O'REILLY®

图灵程序设计丛书



全端Web开发

使用JavaScript与Java

Client-Server Web Apps with JavaScript and Java

[美] Casimir Saternos 著

王群锋 杜欢 译

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

译者介绍



王群锋

毕业于西安电子科技大学，现任职于IBM西安研发中心，从事下一代统计预测软件的开发运维工作。



杜欢

阿里巴巴高级技术专家。2012年加入阿里巴巴，曾就职于雅虎台湾及CISCO。对前端架构、前后端协作有自己的见解，专注于Web产品设计、可用性实施，热爱标准化。

TURING 图灵程序设计丛书

全端Web开发

使用JavaScript与Java

Client-Server Web Apps with JavaScript and Java

[美] Casimir Saternos 著

王群锋　杜欢 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社

北　京

图书在版编目（C I P）数据

全端Web开发 : 使用JavaScript与Java / (美) 萨特诺斯（Saternos,C.）著 ; 王群锋, 杜欢译. -- 北京 : 人民邮电出版社, 2015.8
（图灵程序设计丛书）
ISBN 978-7-115-39730-0

Ⅰ. ①全… Ⅱ. ①萨… ②王… ③杜… Ⅲ. ①JAVA语言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2015)第144108号

内 容 提 要

JavaScript 和 Java 这两大生态系统之间如何协同，成为所有 Web 开发人员共同面临的问题。本书应运而生，全面又简练地为读者展示了最新的 C/S 应用开发范式。本书以 Java 和 JavaScript 这两种最流行的服务器与客户端开发环境为例，全面讲解了最新的 C/S 应用开发范式。作者不仅讲解了很多实用的 C/S 开发架构，还通过各种实例进一步强化了读者的认知。

这是一本写给 Java 程序员的完整的最新 C/S 应用开发范式的指南。

◆ 著　　　[美] Casimir Saternos
译　　　王群锋　杜　欢
责任编辑　毛倩倩
执行编辑　毛倩倩　岳新欣
责任印制　杨林杰

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　印刷

◆ 开本：800×1000　1/16
印张：13.75
字数：326千字　　　　2015年 8 月第 1 版
印数：1－3 500册　　　2015年 8 月北京第 1次印刷
著作权合同登记号　图字：01-2014-8385号

定价：59.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

前言

“计算机科学领域只有两大难题：缓存失效和命名。”

——菲尔·卡尔顿

写一本关于缓存失效的书并不是难事，难的是如何给书取个合适的名字。本书的书名代表的是 Web 开发方方面面的变化，这些变化促成了一种新的 Web 应用设计方法的诞生。

诚然，Web 开发的许多方面都可以称为“新”的。不断升级的桌面浏览器、层出不穷的移动终端、一直在演变的编程语言、更快的处理器，还有越来越挑剔的用户以及他们不断增长的、对易用性和交互性的期望，都让开发者枕戈待旦。这些改变使开发者在为项目提供解决方案时必须持续创新。而其中的许多解决方案并不是孤立的，它们有更为深远的含义。

因此，为了响应这些创新，我选择了“客户端 – 服务器端”这一术语，它在许多方面都反映了 Web 开发中所发生的变化。时下流行的其他一些对现代化开发实践的描述不适用于我们的研究领域。这里所讲的 Web 应用开发是有关桌面浏览器的，另一个日渐相关的领域——移动应用开发，则并不包括在内。

人们使用术语*单页面应用*或*单页面界面*，来区分现代 Web 应用和早期的静态网页。这些术语准确地抓住了现代 Web 应用的特征，和早期应用相比，它们更动态，交互性更强。

当然，很多现代的动态网站并不是单页面应用，它们可能由多个页面组成。这些术语强调的是页面——一个应用的*客户端*。对于服务器端的开发，它们未作任何特别声明。还有与高度动态化的页面相关的各种 JavaScript 框架（Angular、Ember 和 Backbone 等），但是这些仍然是客户端的事。我希望本书包含的内容更广，不局限于*前端*的创新，还应涵盖服务器端的相关设计和 Web 服务信息传递。

这就是流行的 REST（Representational State Transfer）技术，它建议了一种 Web 服务消息传递的风格。但是 REST 的作者罗伊 · T. 菲尔丁（Roy Fielding）只给出了有限的定义，在个

人博客中，他列出了自称为“RESTful APIs”、却常常违反 REST 风格的几个错误（http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven）。还有人在 StackOverflow 上发帖提问 JSON API 是否是真正的 REST 式（http://stackoverflow.com/questions/9055197/splitting-hairs-with-rest-does-a-standard-json-rest-api-violate-hateoas），因为 JSON 并不完全满足 REST 结构风格的约束。对于 REST 服务，有一种渐进式描述（http://martinfowler.com/articles/richardsonMaturityModel.html），即只有当 API 满足相应程度的约束时才可称为 REST 式。REST 将客户端 – 服务器端架构、动词的使用和 URL 的命名规范也作为其约束。

使用 JavaScript 客户端处理实用的“REST 式”API 产生的消息是开发方法中重要的一环，那么服务器端的情况是怎么样的？

Java 企业版（JEE）包含了 JAX-RS API（http://docs.oracle.com/javaee/6/tutorial/doc/giepu.html），它是一种 Java 风格的 REST（但没有严格遵循 REST 规范），并使用了 Jersey 作为一种参考实现予以展示。如果仅限于使用 JAX-RS 开发 Web 应用，则会忽略一些现有的框架和其他一些基于 JVM 的编程语言所提供的解决方案（尤其擅长快速搭建原型）。

为一本书起一个简单明了、引人入胜的书名并不容易，多亏了 James Ward（www.jamesward.com），他在 2012 年的 OSCON 大会上发表了一篇题为“Client-Server Web Applications with HTML5 and Java”的演讲。在这篇演讲中，他列举了一种正在变得流行的 Web 应用开发方法的优点，在近些年的一些项目中，我也使用了这种方法开发 Web 应用。其中，术语“客户端 – 服务器端”是理解这种开发方法的关键，它概括了软件架构的根本性改变，对客户端和服务器端做出了明确区分，并将两者放在了同样重要的位置上。

Web 应用的这种客户端 – 服务器端架构带来了程序员工作方式的转变（有时候，甚至是天翻地覆的变化）。本书就是为了帮助开发者适应这种革命而写的，特别是对于如何构建现代 Web 应用的最新方法方面，也做出了恰当的评价。

目标读者

本书的目标读者是那些熟悉 Java、HTML、JavaScript 和 CSS 的 Web 应用开发者。它是为那些喜欢在实践中学习，喜欢将新技术集成进标准工具中构建示例应用的读者而量身定制的。如果你想了解 JavaScript 的最新发展，以及和使用 Java 在开发流程上的异同，那么请阅读本书。

阅读本书，你需要在两方面之间做平衡。一方面，读者从本书中能得到的最大收获是从宏观上掌握技术转变的影响和趋势；另一方面，理解技术的最好途径往往是具体的例子。如果你的兴趣在于从宏观上掌握各种技术之间是如何融合在一起，那么定能从本书中受益。

我写作此书的目的，就是帮助读者做出明智的决定。好的决定可以帮助你在新项目中采用正确的技术，避免因采用不相容的技术造成的陷阱，或者对给出的决定形成错误的预期。

它还能帮助你在正开发的项目里快速上手，更好地支持已有代码。简言之，明智的决定可以让程序员的工作效率更高，它能让你在当前，或者是未来工作中研究感兴趣的领域时高效地利用时间。

本书内容

第 1 章概览了客户端 – 服务器端的 Web 应用架构，介绍了 Web 应用开发的历史，并为 Web 开发范式的迁移提供了充足的证据。由此引出了接下来的三章内容，分别描述了开发过程中各种工具的使用。

第 2 章介绍了 JavaScript 语言，以及使用 JavaScript 进行开发时用到的工具。

第 3 章介绍了如何设计 Web API、REST，以及开发基于 HTTP 的 REST 式应用时用到的工具。

第 4 章介绍了 Java，以及本书剩余章节内容需要用到的软件。

本书的第二部分探讨了更高级的结构（比如客户端类库和应用服务器），以及它们如何做到良好的分离，从而方便快速开发。

第 5 章介绍了主流的客户端 JavaScript 框架。

第 6 章介绍了 Java API 服务器和服务。

第 7 章介绍了快速开发实践。

第 8 章深入讨论了 API 设计。

对这些类库和快速开发原型的流程有所了解之后，在接下来的几章中，我们将这些知识应用于具体的项目，使用各种基于 JVM 的语言和框架。在接下来的两章中，我们使用了轻量级的 Web 服务器和微框架，而没有采用传统的 Java Web 应用所需的打包方式和服务器。

第 9 章介绍了一个使用 jQuery 和 Jython 的项目。

第 10 章介绍了一个使用 JRuby 和 Angular 开发的项目。

本书的最后几章详细介绍了几个使用传统的 Java Web 应用服务器和类库的项目。

第 11 章介绍了 Java 生态系统中现有的 Web 应用打包方式和部署选项。

第 12 章介绍了虚拟化和在管理大规模服务器环境中涌现出的新技术。

第 13 章将焦点放在了测试和文档化上。

第 14 章对全书做总结，并对互联网相关技术和软件开发中各种纷繁的变化给出了看法。

附录 A 介绍了如何方便地与 Java 类交互。

排版约定

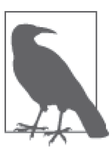
本书使用的排版约定如下所示。

- 楷体
  表示新的术语。

- 等宽字体（`Constant width`）
  表示程序片段，也用于在正文中表示程序中使用的变量、方法名、命令行代码等元素。

- 等宽粗体（**`Constant width bold`**）
  表示示例中的新代码。

- 等宽斜体（*`Constant width italic`*）
  表示应该由用户输入的值或根据上下文确定的值替换的文本。



这个图标表示提示、建议或重要说明。



这个图标表示警告或提醒。

使用代码示例

你可以在这里下载本书随附的资料（项目、代码示例等）：https://github.com/javajavascript/client-server-web-apps。你可以在线查看，或者下载其压缩文件以在本地使用。其中资源是按章划分的。

本书示例代码仅作展示特定功能之用，不求全面展示一个功能完整的应用，具体区别如下。

产品级的系统要对数据类型、验证规则、异常处理例程和日志做很大的改进。

大多数产品级系统都会在后台包含若干个数据存储。为了限制讨论范围，大多数示例都未牵扯数据库。

现代 Web 应用有很大一部分面向移动设备，而且要考虑浏览器兼容性。除非讨论到这些主题，否则我们会避免响应式设计（http://en.wikipedia.org/wiki/Responsive_web_design），而只提供一个最简化的设计。

不可见 JavaScript（unobtrusive JavaScript，参见 http://en.wikipedia.org/wiki/Unobtrusive_JavaScript）将 CSS、JavaScript 和 HTML 分开，是一种被广泛接受的最佳实践。然而在本书的示例代码中，它们常常被混在一起，这是因为对一个应用来说，仅从一个文件就可窥视其奥秘。

单元测试和测试代码示例只在和所讨论主题直接相关时才被包括进来。产品级的系统往往包含大量的测试。

让本书助你一臂之力。也许你需要在自己的程序或文档中用到本书中的代码，但除非大篇幅地使用，否则不必与我们联系取得授权。例如，用本书中的几段代码编写程序无需请求许可，但是销售或者发布 O'Reilly 图书中代码的光盘则必须事先获得授权；引用书中内容或代码来回答问题也无需获得授权，但将大量示例代码整合到你自己的产品文档中则必须经过许可。

使用我们的代码时，希望你能标明它的出处，但我们并不强求。出处信息一般包括书名、作者、出版商和书号，例如：*Client-Server Web Apps with JavaScript and Java*, Casimir Saternos (O'Reilly). Copyright 2014 EzGraphs, LLC., 978-1-449-36933-0。

如果还有关于使用代码的未尽事宜，请随时与我们联系：permissions@oreilly.com。

对长命令的格式化

对于长命令，我们要做适当的调整以方便大家阅读。在操作系统里有一个约定俗成的规则，即使用反斜杠插入新行。比如，下面的两个命令是等价的，在 bash 中执行结果是一样的。（bash 是一种标准的操作系统 shell，当你使用命令行访问 Linux 服务器或 Mac OS X 系统时就会用到。）

```
ls -l *someVeryLongName*
...
ls -l \
*someVeryLongName*
```

这种规则在其他用到操作系统命令的环境下也适用，比如 Dockerfiles。

同样，合法的 JSON 字符串也可以分成多行：

```
o={"name": "really long string here and includes many words"}

// 如大家期望的那样，下面的语句结果为真
JSON.stringify(o)=='{"name":"really long string here and includes many words"}'
```

```
// 同样的字符串，分成多行的结果是一样的，因此下面的结果也为真
JSON.stringify(o)=='{"name":' +
                   '"some really long ' +
                   'JSON string is here' +
                   ' and includes many, many words"}'
```

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的意见和疑问发送给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://oreil.ly/client-server-web-apps-js

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com。

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

感谢人员名单如下：

- Meg、Ally、Simon 和 O’Reilly 公司的朋友们，感谢你们让我有幸写作此书。
- 感谢我的兄弟 Neal Saternos 和 James Femister 博士，他们在我学习编程早期给我很多中肯的建议。
- 感谢 Michael Bellomo、Don Deasey 和 Scott Miller 花费了宝贵的时间审阅本书。
- 感谢 Charles Leo Saternos 从使用 Lua 开发游戏中抽身出来，帮我绘制了一些图形并完成了设计工作。
- 感谢 Caleb Lewis Saternos，从他身上我学到了什么是坚持（有人坚持晨跑吗？），他完成了本书的编辑工作。
- 感谢 David Amidon 给了我从　事软件开发的机会，感谢 Doug Pelletier 让我第一次开发了一个基于 Java 的 Web 应用。
- 感谢那些在项目中并肩作战的兄弟们，他们是经理 Wayne Hefner、Tony Powell、Dave Berry、Jay Colson 和 Pat Doran，当然还有首席软件架构师 Eric Fedok 和 Michael Bellomo。
- 感谢 PluralSight（http://pluralsight.com/training）的 Geoffrey Grosenbach、Webucator（http://www.webucator.com/）的 Nat Dunn，还有 Caroline Kvitka（和来自 Oracle 和 *Java Magazine* 的朋友），感谢他们在过去几年中给了我技术写作的机会，正是因此我才得以在今天写出本书。
- 感谢我的父母 Leo 和 Clara Saternos 在温馨的环境中将我抚养成人，感谢他们在家用 PC 还很罕见时为我购置了一台 Radio Shack Color Computer。还要感谢我的妹妹 Lori，她时刻提醒我生活中还有很多重要的事是和计算机编程无关的。

将我诚挚的爱和谢意献给我最棒的妻子 Christina 和孩子们：Clara Jean、Charles Leo、Caleb Lewis 和 Charlotte Olivia。在本书写作过程中，他们给予了我无尽的爱与支持，他们总是很有耐心，不断给我鼓励。

最后，巴赫在各个层面上都给予了我灵感。特别是他在开始工作时的专注，我愿和他一同呐喊：荣耀归于上帝（http://en.wikipedia.org/wiki/Soli_Deo_gloria）！

第 1 章
因变而变

“企业家总在寻求变化，他们适应变化，并把它当作一次机遇。”

——彼得·德鲁克，“现代管理学之父”

是怎样的变化促使开发者采用客户端 – 服务器端的架构？用户行为、技术和软件开发流程的变迁是其中最重要的原因，它们促使开发者改变过去的设计模式。这其中的每个因素，都在以自己独特且有效的方式让已有的模式不合时宜。它们一起激励了相关领域的创新，虽然没有强制标准化，但是在实践中，人们的开发方式正在趋同。

用户变了。早期的 Web 用户满足于静态页面和最基本的用户界面，而现在的用户期待的是高性能、可交互、精心设计的动态体验。大量新技术的涌现和浏览器能力的扩展满足了用户的期待。现在的 Web 开发者需要使用工具和新的开发方式，以适应现代 Web 应用的使用场景。

技术变了。浏览器和 JavaScript 引擎变得更快。台式机和笔记本性能更强，更不用说无数的移动设备现在也用来网上冲浪了。Web 服务 API 再也不是一个可有可无的功能，而是人们对现代 Web 应用一个再也正常不过的期望。云计算彻底颠覆了 Web 应用的部署和运作。

软件开发方式变了。流行的“敏捷宣言”提倡：

- 个体和互动高于流程和工具；
- 工作的软件高于详尽的文档；
- 客户合作高于合同谈判；
- 响应变化高于遵循计划。

现在，让 Web 应用先快跑起来变成可能——至少在小范围内，可以验证技术的可行性。搭建原型价值非凡。正如《人月神话》（*The Mythical Man Month*，Addison-Wesley Professional）的作者 Fred Brooks 那句名言所说："准备着扔掉一个吧，无论如何，你都会这样干的。"原型让早期客户或用户参与交互，帮助在前期确定需求。花几分钟时间编写一个能工作的 Web 应用已经不是什么不可能的事了。

1.1　Web用户

对如何与现代 Web 应用交互，用户的需求是明确的：

- Web 应用需能跨平台访问；
- Web 应用需在不同平台上提供一致的用户体验；
- Web 应用需响应及时，没有延迟。

高德纳集团（http://www.gartner.com/newsroom/id/1947315）声称，个人云服务将在 2014 年取代 PC 机成为人们数字生活的中心。这对 Web 应用的开发来说有多层含义。用户更加热衷科技，对网站的响应速度有更高的期待。他们不像过去那些年一样被动接受，转而参与其中并互动。网站需要设计得看不出浏览器的局限，提供像原生应用一样的用户体验。

用户希望应用以多种方式提供服务，在不同环境下均可使用。响应式设计、多浏览器、多平台和多设备支持已经成为新的常态。为了支持多样的客户端，使用 JavaScript 类库和框架至关重要。

《纽约时报》最近报道了 Web 用户对网站响应时间的忍耐度（http://www.nytimes.com/2012/03/01/technology/impatient-web-users-flee-slow-loading-sites.html?pagewanted=all&_r=0）。他们发现：如果某公司网站的访问速度比直接竞争对手的慢 250 毫秒，那么访问量就会相对少很多。在开发 Web 应用时，我们需要将性能视作一个关键的考虑因素。

1.2　技术

使用 Java 开发 Web 应用的开发者，一般都熟悉服务器端的动态内容生成。J2EE 和 JSP 经过完善，变成了 JEE 和 JSF。诸如 Spring 这样的框架为服务器端开发提供了额外的功能。在 Web 应用的早期，页面相对来说更静态化，服务器的速度相对更快，JavaScript 引擎很慢，处理浏览器兼容性的类库和技术也很少，这种开发模型就是合情合理的。

与之相比，现在的客户端 – 服务器端架构里，服务器更大程度上负责响应客户端请求，提供资源的访问方式（通常使用 XML 或者 JSON 交换信息）。在过去的服务器驱动模型里，页面（和与之相关的数据）都在服务器端生成完毕，一起返回客户端在浏览器里渲染。而在客户端 – 服务器端架构里，服务器先返回一个包含少量数据的初始页面。用户

和页面交互时，页面向服务器端发起异步请求，服务器端则返回消息，以使页面刷新来响应这些事件。

刚开始的 Web 开发主要是创建静态 HTML 网站，之后加入了服务器端处理（CGI、Java Servlets），为网站注入了动态内容。慢慢地，人们开始使用更加结构化的语言，集成了服务器端模板（ASP、PHP、JSP）和 MVC 框架。新近出现的技术继续沿着传统的路子，增加了这样那样的抽象。

为了向开发者屏蔽设计和 Web 的底层架构，组件化的框架出现了。先是出现了标签库，然后是组件化，几种流行的框架都采用了组件化的方式：

- Java Server Faces(JSF)是一种基于XML的模板系统和组件化框架,导航实行集中化配置；
- Google Web Toolkit 是另一种组件化框架，它发挥了 Java 程序员的长处，让他们集中精力编写 Java 代码，而不需要直接修改 HTML、CSS 和 JavaScript。

每一种框架都有它的作用，被成功应用于生产系统。但是，和很多试图屏蔽底层复杂性的方案一样，当你需要更多的控制权（比如想要集成大量的 JavaScript 代码），或者不满足使用框架的前提条件（比如可用服务器会话）时，就出问题了。Web 的基本架构是使用 HTTP 请求 – 响应协议的一种客户端 – 服务器端计算模型，而这些方案却试图去屏蔽它。

浏览器端的创新也促成了责任从服务器端向客户端的转移。20 世纪 90 年代后期，微软开发了后来成为 Ajax（Jesse James Garrett 于 2005 年 2 月 18 日命名了这一术语）的底层技术。Ajax 是“asynchronous JavaScript and XML”的缩写，但是对各种 Web 页面和服务器进行通信的技术也都适用。这种技术允许少量信息传递，这样在设计基于 JavaScript 的 Web 应用时，可以更好地利用带宽。处理器的升级和 JavaScript 引擎的优化显著地提升了浏览器性能，因此将更多工作从服务器端搬到浏览器端也变得顺理成章。用户界面的响应速度被带向了一个新高度。

移动设备的浏览器更进一步促使了客户端代码和服务器端的分离。有时候，我们可以创建出设计良好的响应式 Web 应用。如果不行，对所有的客户端设备都使用一致的 API 这一点也是很吸引人的。

罗伊·T. 菲尔丁于 2000 年发表的博士论文，让 Java EE 6 与过去基于组件的 API 设计背道而驰。人们设计出了 JAX-RS（Java API for RESTful Web Services）和 Jersey（一个“产品级的参考实现”），用来创建使用 REST 式风格通信的客户端 – 服务器端架构。

1.3 软件开发

过去，建立一个新的 Java 项目相当费事。大量的配置选项让人不胜其烦，而且容易出错。几乎没有自动化，人们假设每个项目都不一样，开发者各有各需要满足的需求。

后来的一些创新，让建立项目变得简单很多。“约定优于配置”是来自 Ruby on Rails 社区的箴言。Maven（http://www.bit.ly/MLOLbU）和其他一些 Java 项目也选择了有意义的默认配置，并为一些常用案例提供了快速创建方式。

JVM 之上出现的各种脚本语言绕过了 Java 严格的类型检查，加快了开发过程。Groovy、Python（Jython）和 Ruby 都是弱类型语言，完成同样的功能需要的代码量更少。还有称为微框架的 Sinatra 和 Play，它们提供了最小化的领域专用语言（DSL），用来快速编写 Web 应用和服务。时至今日，在开发环境中建立一个最小化的 Web 服务已经不是什么难事。

采用瀑布式开发的大型软件项目失败的案例已经不胜枚举，这充分说明为最终产品搭建一个小型原型有很多优势。搭建原型有以下几个目的：

- 验证项目的技术基础；
- 为不同技术搭建桥梁，将它们纳入同一结构；
- 让最终用户和系统交互，明确系统用途和用户界面设计；
- 让系统设计师明确接口和系统之间传递信息的数据结构；
- 让程序员能在应用的各个部分并行工作。

原型还有很多优点，如下。

- 原型是具体的、实实在在的，它们代表了待设计的最终系统。因此，原型融入了本应存在于设计文档、图表和其他介质（常常散见于更随意的场合，比如电子邮件，或人们在饮水机旁的闲谈）的信息。
- 原型是具体的实现。因此，它们代表了真实的需求。这便于人们更好地理解收集上来的需求，并且确定哪些地方需要进一步明确。
- 原型能迅速暴露可能失败的地方。在具体实现前，失败并不是显而易见的。
- 上述优点能更好地帮助人们理解究竟要做什么，这样就会有更好的预测和计划。

由于客户端和服务器端分工明确，在搭建原型开发客户端 – 服务器端的 Web 应用时可以发挥很大的作用。开发客户端的可以使用服务器端的原型（反之亦然），这样就可以并行开发。即使不并行开发，这样也能快速模拟对服务器端的调用，方便开发客户端代码。

1.4　哪些没变

Web 的本质没有变（还是基于 HTTP 进行消息传递的客户端 – 服务器端架构）。

新技术并没有改变一切。高级编程语言没有抹去了解操作系统的需要；对象 – 关系映射框架也没有抹去了解关系型数据库和 SQL 的需要。同样地，在开发 Web 应用时，人们一直试图效仿桌面应用，想忽略 Web 的底层架构。

介质特殊性

介质特殊性是出现在美学与当代艺术批评中的词汇，但是对技术也适用。它隐含了对于给定的艺术主题，需要使用特定介质来表达的合理性。这个观点由来已久，戈特霍尔德·埃夫莱姆·莱辛在《拉奥孔》（*Laocoon: An Essay Upon the Limits of Painting and Poetry*）一书中这样写道：

> 物体连同它们看得见的属性是绘画所特有的题材，动作是诗所特有的题材。[1]
>
> ——论诗与画的界限[2]

当代艺术作品通常挑战艺术上的传统限制。技术是一项创新性活动，但我们关注的是可用的系统，而不是抽象的美。介质无关性之所以重要，是因为如果忽略了平台的本质，则最终系统要么不能以最优的方式工作，要么根本不能工作。这在很多技术领域已经是显而易见的事。本书的目的是提倡遵循 Web 本身的设计方式来设计 Web 应用。由于它遵循了 Web 的基本限制，而不是去忽略它，这样的 Web 应用才能更好地工作。

1 节选自朱光潜的《拉奥孔》译本。——译者注

2 “论诗与画的界限”是《拉奥孔》译本的副标题。——译者注

1.4.1 Web的本质

Web 的本质没有变。它依然由服务器和客户端构成，通过 HTTP 协议，服务器向客户端提供 HTML 文档，如图 1-1 所示。

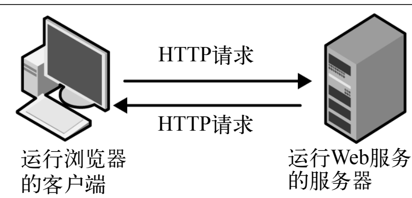


图 1-1：HTTP 请求和响应

客户端 – 服务器端的 Web 应用架构更贴合 Web 本身的架构。虽然与协议无关，REST 是基于 HTTP 开发的，也和 HTTP 结合在一起使用。从本质上说，REST 定义了使用 HTTP 的限制。它试图描述一种设计良好的 Web 应用：该应用是可靠的，运行良好，可扩展，设计优雅，而且方便修改（如图 1-2 所示）。



图 1-2：REST 请求和响应

事实上，为了更准确地强调现代 Web 环境中的挑战，我们还要考虑多设备和云部署，如图 1-3 所示。

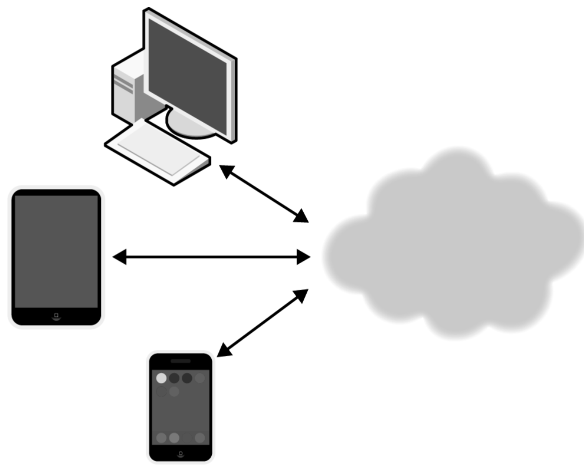


图 1-3：多设备和云部署

在 Web 开发（尤其是组件框架）中，Web 的无状态、客户端 – 服务器端结构的本质，成了被忽略的“介质特殊性”。

1.4.2　为什么说服务器驱动的Web开发有害

具备某种功能，不意味着要使用该功能。在很多情况下，服务器驱动、基于组件的 Web 开发方式都应被客户端 – 服务器端的架构替换。服务器驱动的方式模糊了 Web 的本质，它本身就是一种架构于 HTTP 协议之上的客户端 – 服务器端模型。忽略或模糊 Web 的本质，会让开发、调试和支持软件系统变得更难。服务器驱动的 Web 原本是为了让 Web 变得更容易理解，但这一意愿很快在所有正式的系统（需要对其具备的功能和工作方式有清晰理解的系统）里被打破。

被认为是有害的

1968 年，Edsger W. Dijkstra 发表了一封题为“Go To 语句被认为是有害的”（Go To Statement conidered Harmful）的公开信。这封信很有趣，除了对结构式编程中减少使用 `goto` 语句造成很大影响，它还把“被认为是有害的”（considered harmful）这一说法带入到极客文化中。Tom Christiansen 认为使用 csh 编程是有害的（http://harmful.cat-v.org/software/csh），Douglas Crawford（道格拉斯·克罗克福德）发表了一篇题为“with 语句是有害的”（“with Statement Considered Considered Harmful”，参见 http://yuiblog.com/blog/2006/04/11/with-statement-considered-harmful/）的博文。该说法还见于

其他地方，包括 Eric A. Meyer 发表的那篇风趣的自我指涉的文章“‘被认为是有害的’系列文章是有害的”（“‘Considered Harmful’ Essays Considered Harmful”，参见 http://meyerweb.com/eric/comment/chech.html），而且可以想见，这个说法还会不停地出现。

尽管“被认为是有害的”这类博人眼球的文章质量上参差不齐，但其都有一个合理的共识：具有一种语言特性或一个技术解决方案，不过这并不表示它是好的、长久的解决方案。

1.5 为什么需要客户端–服务器端的Web应用

在 Web 开发中，客户端 – 服务器端的架构有很多优点。

1.5.1 代码组织结构/软件架构

将代码按逻辑解耦的好处很明显，它增强了原有结构和后续支持系统的内聚性。将客户端和服务器端分层，这可以让代码变得可管理、模块化。另外，数据和显示标记能更清晰地分离。可以使用 JSON 而不是内嵌的方式发布数据，这和现代 JavaScript 的不可见 JavaScript 概念是一致的，页面的行为、结构和表现分离。

好的代码组织结构自然会带来灵活性和代码重用。在应用的整个生命周期内，灵活性表现在很多阶段，不同的代码可以分开开发（暴露新的 API、创建移动客户端、测试和发布应用的新版本，这些都可以是独立的）。有了清晰的组件，我们才能更好地重用代码。至少，同一份 RESTful API 可供不同的浏览器和移动设备使用。

组件化的方式易于引入脆弱的耦合，适配性也不好，很难插入不同的前端。

1.5.2 “设计的灵活性”与“使用开源API”

组件化的方式包括了高度集成的服务器端代码，这些代码要求了解特定的 JavaScript 技术。从设计和行为的角度看，生成的 HTML 和 CSS 限制了选择性。一个单独运行 JavaScript 的客户端能使用最新的类库，简化浏览器兼容性、标准化 DOM 操作，并提供复杂的小组件。

1.5.3 原型

由于分层清晰，为客户端 – 服务器端的 Web 应用搭建原型很方便。正如前面提到的，原型可以测试和验证初始想法，帮助澄清模糊的概念，方便清晰地交流需求。用户和这种更具体的东西交互能产生新的想法，这远比冗长的文字描述或图片有用得多。原型还能帮助用户快速地认识，纠正错误的想法和不一致性。如果正确使用，原型能帮助节省时间、金钱和资源，最终产生一个更好的产品。

1.5.4 开发者的效率

除了可以分别（或同时）搭建客户端和服务器端的原型，工作也能清晰地拆分，从而进行并行开发。这种隔离让代码能分开编写，这就避免了模块化方式中页面一更改就得构建整个服务器端代码的问题。开发任务花的时间和精力变少，更改也没有原来复杂，故障诊断也变得简单了。

当需要替换、升级或重新分配服务器端代码时，这点变得尤其明显。这样的改动可以独立完成，不会影响客户端。唯一的限制是原来的接口（特别是 URL 和数据结构）需要保持可用。

1.5.5 应用性能

页面的感知性能对用户体验影响极大。更快的 JavaScript 引擎让客户端可执行计算密集型操作，从而将服务器的工作负荷转移到客户端。Ajax 技术能够按需请求较少的数据，这避免了对整个页面的频繁刷新，请求中传输的数据也减少了。用户和应用交互时，反应更及时，体验更流畅。

无状态的设计让开发者和技术支持的工作更轻松。用于管理会话的资源可以得到释放，这简化了负载均衡和配置。服务器很容易被加进来以应对增长的负荷，方便了*水平扩展*。而且，这还取代了传统的、通过升级硬件提高吞吐量和性能的流程，那种流程真让人头疼。

优点还不止于此，它从整体上简化了系统架构。比如，在云环境中维持状态是件非常有挑战性的事。如果使用传统的有状态会话，高效地持久化数据是个问题，如何才能在一个用户会话的多个请求中保持数据的一致？如果数据存储在后台的一个服务器中，后续被分配到其他服务器的请求就无法访问该数据，可能的解决方案有如下。

使用支持集群和故障转移功能的应用服务器，以 Weblogic 为例，它使用了托管服务器的概念。这种解决方案需要额外的管理工作，每个应用服务器的实现方式也不同。

使用会话关联或*粘滞会话*（sticky session），此时一个用户会话内的所有请求都被发送到同一个后端服务器，但是不提供自动故障转移功能。

使用集中的数据存储。通常都是将数据持久化到数据库中，这种方式的性能不好。

将数据存储在客户端。这避免了将会话数据存储到数据库的性能问题，也解决了使用粘滞会话带来的故障转移问题，因为任何一个后端服务器都能处理来自客户端的请求。

避免管理服务器端状态的做法越来越流行，甚至像 JSF 这样原来被设计成传统的服务器端管理用户会话的框架，现在也加入了支持无状态的功能。

创建客户端 – 服务器端的应用有一些不可避免的挑战。我们有必要将 JavaScript 看作头等

开发语言，这意味着需要深入学习该语言、使用现有类库，以及借鉴成熟的开发技巧。原来被普遍接受的一些架构技术需要重新设计，比如管理会话的标准实践。对客户端 – 服务器端的 Web 应用并没有定义良好的标准。JEE 的某些部分（比如 JAX-RS）澄清了一些概念，但其他框架（如 JSF）并未遵循这些定义。

越过了初期学习（和忘记原有知识）曲线，使用客户端 – 服务器端的模式搭建 Web 应用就变得异常高效和稳定。对服务器端和客户端职责的清晰划分，让修改和扩展代码变得更容易。对于 Web 本质的清楚认识减少了模糊设计带来的问题，水平扩展的能力也大大超出了使用其他设计所能带来的效果。

1.6　小结

新的挑战带来新的机遇。客户端 – 服务器端的架构设计是对 Web 和 Web 开发变化做出的理所应当的响应。它清楚什么没有改变，因而能够指导人们开发出稳定、持久的解决方案，为后续的增强打好了基础。

第2章

JavaScript和JavaScript工具

“JavaScript 是最受轻视的语言，因为它不是其他语言。如果你擅长其他语言，但现在必须在只支持 JavaScript 的环境里工作，那么你必须使用 JavaScript，这真是太烦人了。在这种情况下，大多数人一开始不屑于学习 JavaScript，随后却会惊讶地发现 JavaScript 和他们原本想要使用的其他语言差别如此之大，而且这些差别非常重要。”

——道格拉斯·克罗克福德

道格拉斯·克罗克福德（Douglas Crockford）将 JavaScript 总结为一门很多人使用却鲜有人学习的语言。他写了一本书，列举了该语言的合理用法和强大功能，同时指出了存在问题、需要回避的部分。如果你需要经常使用 JavaScript，应该花时间和精力全面学习。道格拉斯·克罗克福德忽略了该语言中的很多功能，将精力集中在一个强大、简洁的子集，这种方式能帮助程序员更好地学习 JavaScript。

除了学习 JavaScript 语言本身（后来被标准化，称为 ECMAScript），还需要花时间学习特定的编程环境。其他语言运行在操作系统、关系型数据库或宿主应用之上，而 JavaScript 最初被设计成在浏览器上运行。ECMAScript 语言规范（http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf）明确指明了这点。

ECMAScript 最初被设计成一门 Web 脚本语言，提供了一种让浏览器里的页面更加生动的机制，并且将基于 Web 的客户端 – 服务器端架构中一些服务器端的计算任务交给浏览器。

——ECMAScript 语言规范

核心 JavaScript 语言需要结合两个不同的 API 来理解：浏览器对象模型（BOM）和文档对象模型（DOM）。浏览器对象模型包含 `window` 对象及其子对象：`navigator`、`history`、`screen`、`location` 和 `document`。`document` 对象是文档对象模型的根节点，是页面内容结构的一个树状表示。一些对 JavaScript 的抱怨其实是针对浏览器对象模型和文档对象模型的实现问题而言的。不能全面理解这些 API，就不能有效地在 Web 浏览器里进行 JavaScript 开发。

本章剩余部分介绍了在浏览器里使用 JavaScript 进行开发需要了解的主要内容。这个介绍不够全面，只是强调了读者想要深入了解这门语言所需掌握的入手点和知识点。

2.1　学习JavaScript

教学中广泛采用了 Java 作为编程语言，和其有关的认证也已经存在了很多年，因此和 Java 相关的知识已经被充分理解、标准化，成为通识了。人们经常在学校里就学过 Java，工作后又经过自学取得了相关认证。同样的情况并没有发生在 JavaScript 上，但还是有一些关于 JavaScript 的好书可以帮助大家学习的。

JavaScript: The Good Parts（http://shop.oreilly.com/product/9780596517748.do，O'Reilly 出版），道格拉斯·克罗克福德著，该书在前面已经提到过。在圈子里，就某些问题对道格拉斯·克罗克福德提出异议已经成为一种时尚，那是因为他是公认的权威，他帮助很多 JavaScript 开发者形成了自己的思想。有时候，他提出一些过于严格的"法则"，但是如果你不了解 JavaScript 语言的子集（他认为属于"好的部分"）和他尽力避免使用的部分，那就是自讨苦吃。

Secrets of the JavaScript Ninja（http://www.amazon.com/Secrets-JavaScript-Ninja-John-Resig/dp/193398869X，Manning Publications 出版），John Resig、Bear Bibeault 著。John Resig 是 jQuery 之父，他对现实中和浏览器兼容性、操作 DOM 相关的挑战有着深刻的理解。

还有一些书类似于标准语言的参考手册，包括 *JavaScript: The Definitive Guide*（http://shop.oreilly.com/product/9780596805531.do，O'Reilly 出版）、Professional JavaScript for Web Development（http://www.amazon.com/Professional-JavaScript-Developers-Nicholas-Zakas/dp/1118026691，Wrox Press 出版，Nicholas C. Zakas 著）。它们比前面两本书内容更全面（作者的个人观点也少）。它们可能不是那种需要从头读到尾的书，但是在深入某个具体的主题时却是非常有用的。

本节不打算重复你在上述书或其他书中所能学到的全部内容，而是帮助你上手，评估自己的 JavaScript 知识。本章还会引用其他书籍和资源，如果你碰到想要深入研究的术语或概念，可参考它们。

自觉阶段

学习过程中的关键一环是知道自己知道什么，以及知道自己能够知道什么。达克效应（http://en.wikipedia.org/wiki/Dunning%E2%80%93Kruger_effect）是一种认知上的偏差，它描述了这样一种倾向：低技能的人错误地认为他们的能力高于平均水平。鉴于围绕 JavaScript 的困惑和其被嘲笑为一种“玩具语言”的频率，本节的目标（和自觉阶段学习模型相关；自觉阶段学习模型参见 http://en.wikipedia.org/wiki/Four_stages_of_competence）旨在让读者意识到要学些什么。

2.2 JavaScript的历史

关于 JavaScript 的历史已有详尽的记录，Brendan Eich 在 1995 年用 10 天时间写出了 JavaScript 的初始版本（http://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript）。但如果将 JavaScript 放在计算机科学的历史里来考量，尤其是和现存第二古老的高级程序语言 Lisp 相联系，应该会更有意义。John McCarthy 在 1958 年（比 Fortran 晚一年）发明了 Lisp，这是一种计算机程序的数学表达。Scheme 是 Lisp 两种主要的方言之一。奇怪的是，尽管和其他语言的设计反差强烈，Scheme 却在 JavaScript 的历史里扮演了异常重要的角色。图 2-1 列出了一些影响了 JavaScript 设计的主要语言。

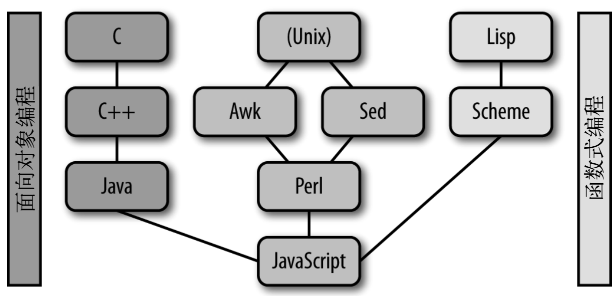


图 2-1：JavaScript 语法继承关系

Scheme 极简主义的设计风格并没有体现在 JavaScript 中，JavaScript 相对冗长的语法来自其他语言，这点在 JavaScript 1.1 规范（http://hepunx.rl.ac.uk/~adye/jsspec11/intro.htm#1006028）中有所提及：

> JavaScript 的语法大多来自 Java，同时继承了 Awk 和 Perl 的一些语法，其基于原型的对象模型间接受到 Self 的影响。
>
> ——JavaScript 1.1 规范

这和 Scheme 截然相反，Scheme 的语法没有受多种语言的影响。Perl 直接影响了 JavaScript 的某些部分，比如对正则表达式的支持。然而 Perl 的箴言：不止一种方法去做一件

事（TMTOWTDI，“there’s more than one way to do it”，参见 http://en.wikipedia.org/wiki/There’s_more_than_one_way_to_do_it）可能在更广的范围上影响了 JavaScript。至少可以反过来说，“只用一种方式去做一件事”（在 Python 社区里很流行）并不适用。请看如下创建和初始化数组的不同方式：

```
var colors1 = [];
colors1[0] = "red";
colors1[1] = "orange";

var colors2 = ["yellow", "green", "blue"];

var colors3 = new Array(2);
colors3[0] = "indigo";
colors3[1] = "violet";

var colors4 = new Array();
colors4[0] = "black";
colors4[1] = "white";
```

因此，看起来 JavaScript（及其受到多种语言的影响和语法上的变种）和 Scheme（Lisp 的极简方言）之间似乎没有任何关联。但是，JavaScript 的确和 Scheme 关系紧密（http://brendaneich.com/tag/history/），直接受其影响：

> 就像我常说的，Netscape 的其他人也可以作证，我受雇于 Netscape 时承诺“在浏览器里使用 Scheme”。
>
> ——Brendan Eich

这一点也反映在 ECMAScript 语言规范（http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf）里：

> ECMAScript 中的一些技术和其他编程语言使用的类似，尤其是 Java、Self 和 Scheme。
>
> ——ECMAScript 语言规范

来自 Scheme 的影响也被其他人辨认出来了。道格拉斯 • 克罗克福德根据 Daniel Paul Friedman 那本经典书 *The Little Schemer*（http://mitpress.mit.edu/books/little-schemer，MIT Press 出版），写了一篇文章“The Little JavaScripter”（http://www.crockford.com/javascript/little.html，列举了 Scheme 和 JavaScript 的共同点。Lisp 社区（欧洲 Lisp 研讨会，参见 http://www.european-lisp-symposium.org/）也将 ECMAScript 描述为一种“Lisp 方言”。JavaScript 和 Scheme 语言之间的相似性不可否认，这是由创造者的本意决定的。

2.3 一门函数式语言

Java 开发者倾向于站在面向对象的角度解决问题。尽管 JavaScript 也支持面向对象，但是

这却不是解决问题最高效的方式。使用 JavaScript 的函数式编程能力会更高效。理解了什么是函数式编程和它的含义，就弄清楚了这门语言的本质和能力。

JavaScript 和 Scheme 语言相像的主要特征是它是一门*函数式编程语言*，这既和它的起源相关，也和它的语法相关。这里的函数式编程语言是指既支持函数式编程（http://en.wikipedia.org/wiki/Functional_programming），又支持将函数当作一级对象（http://en.wikipedia.org/wiki/First-class_function）。JavaScript 的这一基本概念为语言的其他方面提供了方向。对很多程序员，尤其是那些以类似 Java 这样还未直接支持函数式编程的语言为基础的程序员来说，使用函数式编程是非常大的范式迁移[1]。

2.3.1 作用域

作用域是指程序中变量可见和可操作的范围，在 JavaScript 中这个概念让人很难捉摸。像很多其他语言一样，函数可用来包括一组语句。这样就能复用函数，同时将信息可见性限制在一个易理解的模块化单元中。ECMAScript 语言规范（http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf）定义了三个执行上下文：全局、eval 和函数。和其他类 C 语言不同，JavaScript 没有区块级作用域，但是有函数级作用域。像 `if` 语句这样的区块结构不会产生新的作用域。

使用 JavaScript 的危险之一是方法或变量被*提升*到作用域顶端，它们是在那里定义的。函数声明在作用域执行时就已经存在了，所以函数被*提升*到执行上下文的顶端。按照经验，可在作用域顶端使用 `var` 声明所有要在作用域里用到的变量来避免这类问题：

```
//这不是Java中的成员变量……
var x = 'set';

var y = function () {

// 你可能想不到 -> var x; 被提升到这一行!

    if (!x) { // 你可能觉得该变量此时已经被初始化
              // 但是却没有,因此会执行该段代码
        var x = 'hoisted';
    }

    alert(x);
}

//……这条语句会弹出警告,显示"hoisted"
y();
```

注 1：函数式编程在 JVM 上已经存在一段时间了，使用过一些基于 JVM 的脚本语言，包括 Rhino JavaScript 实现。Java 8 计划加入 Lambda 表达式、闭包和相关语言特性。Java 8 还会加入一个新的 JavaScript 实现：Nashorn。随着不断加入新功能，未来几年，JavaScript 开发，尤其是函数式编程将变成 Java 开发者需要深入学习的内容。

这个例子还包含了一些在 Java 中见不到的特性。

- 在 JavaScript 中，`null`、`undefined` 和其他一些值被当作 `false`。
- `if` 语句中的条件判断表达式是 `!x`,感叹号代表逻辑非操作,因此,如果 `x` 是 `undefined`（或者 `null`），则 `if (!x)` 为 `true`。如果 `x` 是一个数字或字符串，则会像使用过其他语言的开发者期望的那样，值为 `false`。
- 使用 `var` 关键字定义局部变量，没有使用该关键字定义的变量为全局变量。使用 `var` 关键字定义的局部变量的作用域和函数的作用域相关。
- 创建一个函数后将其赋值给变量 `y`，这对 Java 程序员来说有些奇怪，因为在他们的世界里，方法只和类或对象的实例关联。该语法展现了 JavaScript 函数式语言的本质。

2.3.2 一级函数

从任何严格的意义上来说，拥有限定作用域的函数并不能归为函数式语言。函数式语言是指支持一级函数的语言。根据 *Structure and Interpretation of Computer Programs*（http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-12.html#%_idx_1218）这本书的描述，一级函数可以赋给一个变量，作为参数传递给另外一个函数，作为函数的返回值，或者包含在其他数据结构中。下面的例子（为说明问题人为设计的）展示了这些功能，还有一些专家认为函数式编程语言必须具备一个特性：支持匿名函数。

```
//
// 可在任何一款现代浏览器的JavaScript控制台里执行下面的程序
//

// 将函数赋给变量
var happy = function(){
    return ':)';
}

var sad = function(){
    return ':(';
}

// 该函数接收一个函数(作为参数),又返回一个函数
var mood = function(aFunction){
        return aFunction
}

// 将函数加入一种数据结构,即数组中
list = [happy, sad]

//……加入JavaScript对象
response = {fine: happy, underTheWeather: sad}

// 传入一个函数,调用后返回另一个函数,并将该函数赋给一个变量
var iAmFeeling = mood(happy);
console.log(iAmFeeling());
```

```
// 再来一次
var iAmFeeling = mood(sad);
console.log(iAmFeeling());

// 函数可以被包含在一个数据结构里
// 这里的数据结构是一个JavaScript对象

console.log(response.fine());

// 或者你想使用数组这种数据结构……
console.log(list[0]());

// 最后,直接定义和使用一个匿名函数
console.log(function(){
        return ";)";
}());
```

从这个例子中可以清楚地看到，函数是 JavaScript 中的基本单元，是名副其实的一级对象。它可以脱离对象和其他结构单独存在，也可以出现在任何表达式可以出现的地方。和 JavaScript 中其他对象的区别是：函数可以被调用。由于函数是一级对象，而且是主要的可执行单元，使用函数可以写出短小精悍的代码。和函数相关的作用域产生了一些习惯用法，很多 JavaScript 新手对此并不熟悉。

JavaScript 真的是函数式的吗？

有些人质疑 JavaScript 是否能称得上是一门函数式编程语言。毕竟，函数式编程是在模仿没有副作用的数学函数。而任何使用过 JavaScript 的人，都见识过其臭名昭著的全局上下文，而且操作 DOM 时，几乎百分之百会用到充满副作用的函数。引用透明性就无从谈起了。而且，很多 JavaScript 程序都知悉周围环境，变量是可变的。纯函数式编程语言使用不可变变量（这带来了很多好处，比如方便实现并发操作）。

JavaScript 有对象和基于原型的继承，因此也可以说它是面向对象的，至少它是一种支持多范型的编程语言。

JavaScript 包含函数，支持将函数作为一级对象，这是无可争辩的事实。读者可自行选择“函数式编程语言”的定义（因为并不存在一个权威的定义），对 JavaScript 是否属于函数式编程语言做出自己的判断。本书使用函数式编程是因为其突出了 JavaScript 中优质的功能。读者如需从这方面更加深入了解 JavaScript，可参考 Michael Fogus 所著的 *Functional JavaScript*（http://shop.oreilly.com/product/0636920028857.do，O’Reilly 出版）一书，该书介绍了很多使用 JavaScript 的函数式编程技巧，其中很多都使用了 underscore.js 类库（http://underscorejs.org/）。

2.3.3　函数声明和表达式

JavaScript 中的字面函数由以下四部分组成：

- function 操作符；
- 可省略的函数名；
- 一对小括号（包含零到多个参数）；
- 一对大括号（包含零到多条语句）。

在 JavaScript 中，一个合法的最小化*函数声明*如下所示：

```
function(){}
```

函数可以有函数名，这和传统的类 C 语言的语法风格比较像：

```
function myFunctionName(){}
```

没有名字的函数称为匿名函数。*匿名函数*可以在一个表达式中使用，可以被赋给一个变量。有些人喜欢这样的语法，因为它能让人清楚地意识到变量保存的是一个函数值：

```
var x = function () {}
```

具名函数也可以赋给一个变量：

```
var x = function y() {}
```

这种使用方式，让函数外部得以用变量 x 引用函数，函数内部也可以通过函数名 y 引用该函数（递归调用）。

函数可以关联至一个对象，这时称之为*方法*。对象被隐式传递给其所调用的方法，方法可以访问和操作对象里的数据，通过 this 关键字引用对象，如下所示：

```
var obj = {}; // 创建一个新的JavaScript对象
obj.myVar = 'data associated with an object'
obj.myFunc= function(){return 'I can access ' + this.myVar;} // this: 引用该对象
console.log(obj.myFunc())
```

可以在函数中定义其他函数，内部的函数可以访问外部函数的变量。一个函数返回一个内部函数时，就形成了*闭包*。返回的对象既包含了函数本身，也包含创建函数时的环境：

```
function outer() {
    var val = "I am in outer space";
    function inner() {
        return val;
    }
    return inner;
}

var alien = outer();
console.log(alien());
```

立即执行函数（immediate function）将代码限定在函数的局部作用域内，避免了污染全局

作用域：

```
(function() {console.log('in an immediate function')}());
```

2.3.4　函数调用

有四种方式调用函数：

- 函数；
- 方法；
- 构造函数；
- 使用 call() 或 apply()。

不同的方式会影响 this 关键字引用的对象。第一种方式（函数），在非严格模式下，this 指全局上下文；在严格模式下，返回 undefined 或者在执行上下文中得到的值。接下来的两种方式（方法和构造函数）是面向对象编程所特有的，其中方法调用是调用和对象关联的函数，而调用构造函数会创建一个新对象。和以上三种方式不同，call() 和 apply() 允许在调用一个函数时，显式指定上下文。

> **this 和 that**
>
> JavaScript 中有个惯例，乍看之下让人莫名其妙：
>
> ```
> var that = this
> ```
>
> 在理解了 JavaScript 中的 this 是如何工作的之后，这种用法就一目了然了。this 随上下文（作用域）而变，所以一些开发者将它赋给 that，以访问 this 原来所指的值。

2.3.5　函数参数

前面提到过，函数通过签名中声明的命名参数接收参数。有一个特殊的变量 arguments，它保存了所有传入函数的参数，不管是有名的还是没名的。下面的例子展示了如何分别使用标准的函数调用、call() 和 apply() 将三个数字相加：

```
function add(){
    var sum=0;
    for (i=0; i< arguments.length; i++){
        sum+=arguments[i];
    }
    return sum;
}

console.log(add(1,2,3));
console.log(add.apply(null, [2,3,4]));
console.log(add.call(null,3,4,5));
```

2.3.6 对象

在 Java 中，对象是所定义的类的实例。在 JavaScript 中，对象只是一组属性的集合。JavaScript 中的对象也有继承（从它的原型那里），面向对象的设计原则对它也是适用的，但是它和 Java 中的方式大相径庭。可以在 JavaScript 中创建类（http://www.phpied.com/3-ways-to-define-a-javascript-class/），但是以 Java 中的方式去思考它是行不通的（Java 中的类是基本的、必需的）。

基于类和基于原型的继承让 JavaScript 和 Java 如此不同，这也导致了很多疑惑。JavaScript 中的其他特性也可以和 Java 对照着来看。

2.4 面向Java开发者的JavaScript

如果仅从表面上看，大多数开发者认为 JavaScript 只不过组装了 Java 或其他类 C 语言的语法（`for` 循环、条件语句等），但是如果看一个完整的 Java 程序，差别马上就出来了。下面是一个经典的 Java 程序，展现了和 JavaScript 代码以及开发方式上的不同。

2.4.1 HelloWord.java

```
/**
 * HelloWorld
 */
class HelloWorld{

        public static void main (String args[]){
                System.out.println("Hello World!");
        }
}
```

想要在命令行里看到这个久负盛名的 Hello World 程序的输出，你必须：

(1) 创建一个名为 HelloWorld.java 的源文件；
(2) 将 Java 代码编译为类文件（在命令行里使用 `javac` 编译器）；
(3) 执行类文件（在命令行里使用 `java` 解释器）。

如果你使用像 Eclipse 或 IntelliJ 这样的集成开发环境，这些步骤都有对应的菜单项。很简单，执行该程序，输出“Hello World!”。但是这个简单的例子却说明了 Java 和 JavaScript 的诸多不同之处。

下面是对应的 JavaScript 程序，输出结果是一样的：

```
console.log('Hello World')
```

1. 程序执行

首先，JavaScript 是一门解释性语言，不需要编译。该在什么环境里执行 JavaScript 代码呢？如果你安装了 node（http://nodejs.org/），可在 node 的命令行里执行：

```
> console.log("Hello World")

Hello World
```

也可以在浏览器控制台里执行。在 Chrome 中，选择视图→开发者→ JavaScript 控制台（如图 2-2 所示）。



图 2-2：Chrome JavaScript 控制台

在 Firefox 中，选择工具→ Web 开发者→ Web 控制台（如图 2-3 所示）。



图 2-3：Firefox JavaScript 控制台

其他一些现代浏览器也提供了类似的功能。

宿主对象

从技术的角度讲，JavaScript 本身没有内置输入 / 输出功能（由运行时环境提供，这里是由浏览器提供）。根据 ECMA 标准（http://www.ecma-international.org/ecma-262/5.1/#sec-4）：

> ECMAScript 并未被定义成一门计算完备的语言。事实上，本标准未定义如何输入外部数据，也未定义如何将计算结果输出。相反，ECMAScript 程序的运行环境不仅负责提供本标准描述的对象和结构，也负责提供和环境相关的宿主对象。对它们的描述超出了本标准的范围，本标准只略微提及它们的一些属性和方法，这些属性和方法可以在 ECMAScript 程序中访问或调用。

标准中未定义这些不仅是因为好奇，微软的 IE 浏览器有好几个版本都没有 console.log，常常会引发一些不可预料的错误。很多和 JavaScript 相关的挑战和问题，其责任都在运行环境（通常是浏览器）。DOM 是一个跨平台、和语言无关的引用及操作 HTML 元素的方法，它本身并不是 JavaScript 语言的一部分。

让我们再回到那个 Hello World 程序。你可能已经注意到，我们使用了单引号，而不是双引号，也没有以分号结尾。JavaScript 的语法更宽容（或者说更含混）。有很多发明来减少因此造成的困惑。语言本身添加了严格模式（https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/Strict_mode）。由道格拉斯·克罗克福德开发的 JSLint（http://www.jslint.com/）等工具强迫程序员使用 JavaScript 语言中"好的部分"，你可参考他写的书（http://www.amazon.com/exec/obidos/ASIN/0596517742/wrrrldwideweb）深入了解这方面内容。可以这样说，JavaScript 有很多陷阱，不守纪律的开发者或团队使用它会产生一些难于调试的问题。花些时间，好好学习该门语言以养成一些习惯和实践来避免这些问题，是很值得做的一件事。

2. 文件系统组织

Java 项目的文件和目录结构与代码的结构是直接相关的。一个源文件通常包含一个（公有）类，类名和文件名相同。文件的目录结构反映了类的包名（对于内部类、访问修饰符和其他一些结构会有例外，但是在 Java 项目里，文件和目录结构通常都遵循类似的结构）。有时候，这些限制会带来不便（特别是对于小型项目）。随着项目的成长，这种方式清晰地表明了代码的组织结构。不用打开文件，只需瞥一眼目录结构就能立即知晓一个项目是否组织混乱。

JavaScript 则没有这种限制，文件或目录结构没有必然的联系。因此，随着项目的成长，你要特别注意代码的组织方式，如果是一个大型开发团队，甚至要花时间来重构代码的组织结构。Alex MacCaw 在 *JavaScript Web Applications*（http://shop.oreilly.com/product/0636920018421.do，O'Reilly 出版）一书中很好地解释了这些：

> 开发大型 JavaScript 应用的秘诀是不要开发大型 JavaScript 应用。你应该将应用分解成一些彼此独立的模块。开发者在开发应用时常犯的错误之一是引入了太多的互相依赖，大量的 JavaScript 代码生成大量的 HTML 标签。这样的应用难以维护和扩展，必须想方设法避免。
>
> ——Alex MacCaw

关于代码组织，还有其他一些方面需要考虑。除了命名文件和将代码保存在合适的文件里，文件之间的依赖需要依次加载。当 JavaScript 代码被放到 Web 服务器上，按需加载可以提升效率（等待所有文件都下载完成会让浏览器僵死）。可使用由 RequireJS（http://requirejs.org/docs/whyamd.html）等类库支持的 AMD（Asynchronous Module Definition，异步模块定义）API（https://github.com/amdjs/amdjs-api/wiki/AMD）来提升性能。该 API 按模块定义 JavaScript 代码，让模块和其依赖能异步加载。

2.4.2 带变量的HelloWord.java

演示 Hello World 程序的下一步通常是用它跟一个变量代表的名字"打声招呼"：

```
/**
 * HelloWorld2
 */
class HelloWorld2 {

    public static void main (String args[]) {
        String name;
        System.out.println("Hello " + name);
    }
}
```

上述代码不能编译：

```
HelloWorld2.java:5: variable name might not have been initialized

var name;
console.log('Hello ' + name);
```

上述 JavaScript 程序可以运行，但它却是让人产生困惑的源头之一：太多的值被当成 `false`。求值的过程充满困惑，难于记忆：

```
// 结果都是false
console.log(false     ? 'true' : 'false');
console.log(0         ? 'true' : 'false');
console.log(NaN       ? 'true' : 'false');
console.log(''        ? 'true' : 'false');
console.log(null      ? 'true' : 'false');
console.log(undefined ? 'true' : 'false');
```

下面这些结果是 `true`：

```
// 结果都是true
console.log('0'       ? 'true' : 'false');
console.log('false'   ? 'true' : 'false');
console.log([]        ? 'true' : 'false');
console.log({}        ? 'true' : 'false');
```

在 Java 中初始化变量后，程序就可以编译并执行了：

```
/**
 * HelloWorld2
 */
class HelloWorld2{

    public static void main (String args[]){
        String name = "Java";
        System.out.println("Hello " + name);
        }

}
```

同样，如果给 JavaScript 中的变量一个初始值，输出会如我们期待的那样：

```
var name='JavaScript';
console.log('Hello ' + name)
```

如果在全局作用域，关键字 var 并不是必需的，去掉它程序行为没有任何不同。如果是在一个函数里调用，var 会创建一个局部变量。一般来说，应该在函数里使用关键字 var 声明变量，防止污染全局命名空间。

在 Java 的例子中，声明变量需要指定类型。JavaScript 是一种弱类型的语言，不需要这样做。typeof 操作符可以显示大多数常用类型信息，如表 2-1 所示。

表2-1：JavaScript中typeof操作符的例子

类型	结果	例子
Undefined	“undefined”	`typeof undefined`
Null	“object”	`typeof null`
Boolean	“boolean”	`typeof true`
Number	“number”	`typeof 123`
String	“string”	`typeof "hello"`
Function object	“function”	`typeof function(){}`

2.5 最佳开发实践

JavaScript 有其自身的挑战和特性，需要特别的开发流程。尽管我们可以死搬硬套熟悉的 Java 开发流程，但使用适合 JavaScript 的工具和流程会更好。

2.5.1 编码规范和约定

本书的大部分内容都关乎如何做到客户端和服务器端的松耦合。不可见 JavaScript 是使客户端 UI 层实现松耦合的一组最佳实践：

- 使用 HTML 定义页面数据结构；
- 使用 CSS 为数据结构增加样式；
- 使用 JavaScript 为页面增加交互功能。

也可以这样说：

- 避免在 CSS 中使用 JavaScript；
- 避免在 JavaScript 中使用 CSS；
- 避免在 HTML 中使用 JavaScript；
- 避免在 JavaScript 中使用 HTML。

2.5.2 浏览器

浏览器无处不在，以致于很多 Web 用户都不清楚它和底层操作系统的区别。浏览器不仅是终端用户浏览网页的环境，它还是个 IDE。浏览器集成了调试器、代码格式化工具、分析工具和许多其他工具（有些以插件和扩展的形式存在），在开发过程中会用到它们。

很长一段时间内，Firefox 和 Firebug（https://getfirebug.com/）等开发插件和扩展是开发中流行的浏览器，Chrome 也自带了一些开发者工具，大有后来者居上之势。

Chrome 小贴士

花点时间研究浏览器提供的开发者工具和影响浏览器行为的命令行选项是值得的。为了开发者访问方便和提高生产效率，经常会越过一些安全限制或牺牲一点性能，比如在 Chrome 中：

- 浏览器有清除缓存（https://groups.google.com/forum/#%21msg/angular/wMRtJZ7R480/5bFH_ZhgdRwJ）等选项，可以防止更改代码时引起的困惑；
- 根据操作系统和浏览器版本的不同，命令行语法也略有差异，比如通过以下命令可以在 OS X 上运行 Chrome：

```
/Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome
```

在命令行里加入一些标签可改变浏览器行为。如果你的文件使用 Ajax（比如使用 AngularJS），`--allow-fileaccess-from-files` 标签可以让你脱离 Web 服务器开发。如果你需要使用 JSONP 引用本机（localhost），则必须使用 `--disable-web-security` 选项。

在 Chrome 中，还有一些隐藏功能。在地址栏里键入 `chrome://chrome-urls/`，会列出所有可用的通往其他配置或管理界面的 URL。为了小试牛刀，键入 `chrome://flags/` 来列出当前版本的浏览器所提供的实验性功能吧。

浏览器作为 JavaScript 的开发环境的另一个例子是许多 JavaScript 开发在线合作网站的出现，比如 JSFiddle（http://jsfiddle.net/）和 jsbin（http://jsbin.com/）。可以在这些网站上创建示例应用以重现缺陷、和其他人以精确的方式交流语言中的某些特定问题。没有理由使用脱离上下文的 JavaScript 代码片段。在线上提问或演示某项技术时，提供一个具体的例子或者实现一个小型的演示程序是很正常的。

2.5.3 集成开发环境

WebStorm（http://www.bit.ly/jb-webstorm）是一款非常优秀的 IDE，它又轻又快，包括了调试器、一个很棒的项目视图、强大的快捷键、通过使用插件的扩展方式和一些其他功能。虽然不是必需的，但 WebStorm 通过使用向导、快捷键和代码补全功能，将很多最佳实践付诸其中。

2.5.4 单元测试

有很多为 JavaScript 开发的单元测试框架（http://www.bit.ly/1gvnROC）。书中用到的 Jasmine（http://pivotal.github.io/jasmine/）是一个行为驱动开发（BDD，http://www.bit.ly/1dkUNKU）的框架，它语法简单，而且没有外部依赖。

Java 的单元测试能在每次构建项目时通过构建工具或脚本运行。JavaScript 的单元测试可使用一个 Node.js 的模块，Karma（https://github.com/karma-runner/karma，原来叫作 Testacular）在多浏览器里执行，并且每当修改并保存源文件后，就会自动执行。这一点很重要。如果你能自律编写单元测试，每当文件保存时就执行单元测试这一能力将能够在早期发现缺陷。这点对 JavaScript 来说尤其有用，因为 JavaScript 没有编译器，不能在早期验证代码的合法性。有效的单元测试常常扮演一个伪编译器的角色，它们能够立刻反馈代码的质量，并且一有缺陷，就能检测到。

2.5.5 文档

JavaScript 中有很多自动化文档生成工具。JSDoc（https://github.com/jsdoc3/jsdoc）有着和 Javadoc 类似的标记和输出，Dox（https://github.com/visionmedia/dox）是一个生成文档的 Node.js 模块。

文学编程（http://en.wikipedia.org/wiki/Literate_programming，由高德纳在 20 世纪 70 年代提出）让程序员用他们自己思考的逻辑和流程顺序开发程序。Docco（http://jashkenas.github.io/docco/）是一个 Node.js 模块，它将代码和注释组织成一种类似文章的格式。虽然 Docco 不直接验证和执行代码规范，使用它却能鼓励大家使用良好的代码结构和注释，而不是不假思索地复制粘贴。

2.6 项目

这是一个小型的展示对象继承关系的 JavaScript 项目，包含了单元测试和文档。所有文件均能通过访问 GitHub（https://github.com/java-javascript/client-server-web-apps/tree/master/Chapter-2-JavaScript-And-Tools/Animals）获得。

`Animal.js` 是所有对象的根对象：

```
// Animal处于对象继承关系的顶部
function Animal() {}

// 定义speak方法,子类中有该方法的不同实现
Animal.prototype.speak = function() {
    return "Animal is speaking.";
};
```

它有两个子类，一个是 Cat.js：

```
// 定义Cat类
function Cat() {
    Animal.call(this);
}

// 设置对象的原型
Cat.prototype = new Animal();

// 使用类名为构造函数命名
Cat.prototype.constructor = Cat;

// 实现具体的函数
Cat.prototype.speak = function(){
    return "meow";
}
```

一个是 Dog.js：

```
// 定义Dog类
function Dog() {
    Animal.call(this); // 调用父对象的构造函数
}

// Dog继承自Animal
Dog.prototype = new Animal();

// 更新构造函数以和新类匹配
Dog.prototype.constructor = Dog;

// 替换speak方法
Dog.prototype.speak = function(){
    return "woof";
}
```

最新版本的 Jasmine 可在 GitHub（https://github.com/pivotal/jasmine）上获得：

```
curl -L \
https://github.com/downloads/pivotal/jasmine/jasmine-standalone-1.3.1.zip \
-o jasmine.zip
```

下面是一个测试了上述定义的每个类的单元测试，可以在 Jasmine 上运行：

```
// 使用beforeEach测试Animal
describe("Animal", function() {

    beforeEach(function() {
         animal = new Animal();

    });

    it("should be able to speak", function() {
```

```
        expect(animal.speak()).toEqual("Animal is speaking.");
    });
});

// Dog继承自Animal,重写了speak方法
// 测试中使用了一个局部变量
describe("Dog", function() {

    it("should be able to speak", function() {
        var dog = new Dog();
        expect(dog.speak()).toEqual("woof");
    });
});

// 还能更简洁一点:Cat继承自Animal
// 在一行内同时调用构造函数和speak方法
describe("Cat", function() {

    it("should be able to speak", function() {
        expect((new Cat()).speak()).toEqual("meow");
    });

});
```

最简单的测试方式是在浏览器里打开 SpecRunner.html，结果如图 2-4 所示。

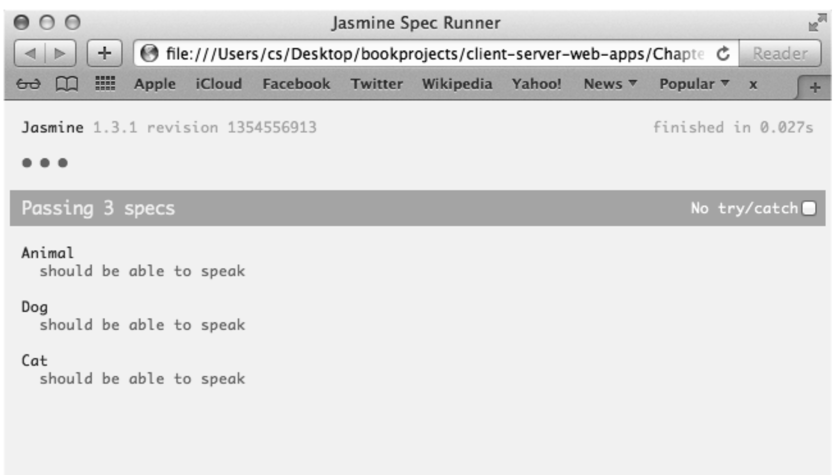


图 2-4：使用 Jasmine 的例子

如果需要在文件每次改动后自动运行测试，需要安装 Node.js（http://nodejs.org/）。可以通过如下查看 Node.js 的命令确认是否成功安装了 Node.js：

```
node --version
v0.8.15
```

还需要安装 Node.js 的包管理工具：

```
npm --version
1.1.66
```

安装了这些后，就可安装 Karma:

```
npm install karma
```

安装成功后，可通过如下命令查看帮助：

```
karma --help
```

项目的配置文件通过 `init` 命令生成，该命令会生成一个 karma.conf.js 文件，后续可以修改此文件，使其指向项目中的 JavaScript 文件，在浏览器里运行。一经配置，使用 `start` 选项，就可以在每次更改文件后自动运行单元测试。

使用 `npm` 安装 `docco` 模块，用以生成文档：

```
npm install docco
```

运行 `docco` 命令会在 docs 文件夹生成 HTML 文档。图 2-5 展示了生成的文档。注释在左，高亮语法显示的代码在右。可以通过右上角的下拉列表选择要显示的其他文件：

```
docco app/*.js
```

安装 Pygments

如果你看见如下信息："Could not use Pygments to highlight the source"，这是指一个叫 Pygments（http://pygments.org/）的使用 Python 编写的语法高亮显示工具。

在 Mac 或 Linux 系统下，使用如下命令安装：

```
sudo easy_install pygments
```

我们快速浏览了 JavaScript 对开发的支持。使用其中的一些工具，能有效提高代码的质量和可维护性。

本章只是概述了这门复杂且普遍使用的语言。对于这里所讲述的内容，其他图书提供了更为深入的讲解。比如由 Alex McCaw 所著的 *Java Script Web Applications*（http://www.bit.ly/js-web-applications?cc=73c675463d90439868abb117faf4f9a2，O'Reilly 出版）一书讲解了构建大型 JavaScript 应用所使用的模式（模型 – 视图 – 控制器）和相关技术。由 Nicholas C. Zakas 所著的 *Maintainable JavaScript*（http://shop.oreilly.com/product/0636920025245.do，O'Reilly 出版）一书讲解了编写可维护、可扩展的 JavaScript 代码的实践和标准。当然，还

有很多在线资源，比如 Mozilla Developer Network（https://developer.mozilla.org/en-US/docs/Web/JavaScript）和 StackOverflow（http://stackoverflow.com/questions/tagged/javascript?sort=frequent&pagesize=15）。

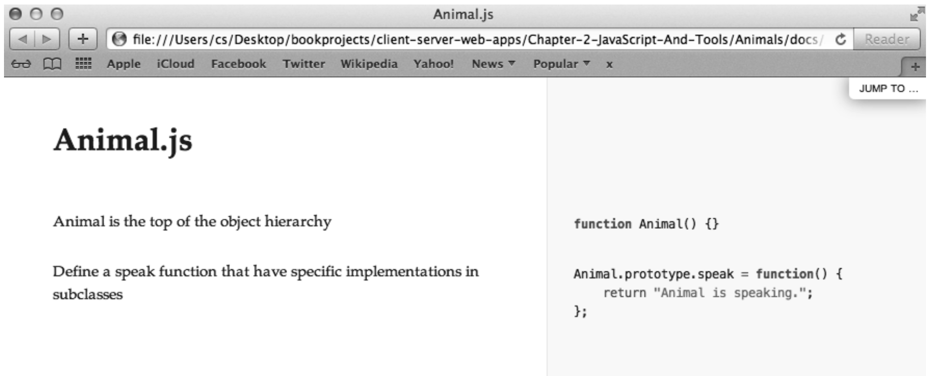


图 2-5：Docco 截图

JavaScript 不是 Java，它有自己迅速发展的生态系统，其开发工作有很多项目支持。了解这一语言和相关开发工具，这会为你探索后续将要介绍的很多框架和类库打下基础。

第3章

REST和JSON

“好篱笆造出好邻家。”

——罗伯特·弗罗斯特

一旦引入一种技术，人们的第一反应是它不能做什么。对那些习惯于在限定范围内工作的人来说，这种反应会让他们感到迷惑。从负面的角度来看“篱笆”，它增加了限制；从正面的角度看，正如罗伯特·弗罗斯特说的那样，它明确了界限、暗示了用意和潜在的危险。同样的道理，软件系统的约束提供了更好的功能、效率，明确了角色。好的约束不在于限制，而是为了达到正向的结果。REST 是一种架构风格，它由一系列约束组成，描述了 Web（或者类似结构）应该如何工作。这些约束虽然增加了限制，但是却孕育了适合 Web 本质的系统设计。

对于 REST 的描述，通常以这样一种观点开始：REST 是一种 Web 服务协议。这样理解没问题，尤其是和 SOAP 或其他消息服务做类比时。和 SOAP 相比，人们认为 REST 是一种简化的协议，缺少实现 SOAP 时必需的扩展定义和额外结构。REST 直接依赖底层 HTTP 协议的功能，包括请求方法、URI 地址和响应代码。REST 也可以和远程过程调用（RPC）风格的 API 来对比，远程过程调用用 URL 表示行动，REST 用 URL 访问资源。这样说来，REST 是面向名词的，而 RPC 是面向动词的。

为什么选择 REST，而不是 SOAP?

关于 REST 和 SOAP 孰好孰坏的辩论不绝于耳。SOAP 提供了规范的约定，适合复杂的 RPC 架构，但对大多数 Web 开发来说，它引入了额外的开销和复杂度，而且也没有明显的好处。SOAP 依然作为一个 RPC 平台被大量使用，但这并非出于技术因素，很多时候只是因为现有的大型系统里大量沿用了 SOAP。

SOAP 消息臃肿，需要大量处理，而且使用烦琐的 XML 信封来表示。这使得客户端 - 服务器端的 Web 应用对该协议提不起兴趣，尽管它可以很容易地使用 JavaScript 处理 JSON。

很多开发者开始接触 REST 时，都将其作为一种拥有整齐的 URL 和自由轻量消息的 Web 服务协议。从这个角度理解 REST 自有其实用价值，但以其作者罗伊 • T. 菲尔丁（Roy T. Fielding）的表述来理解 REST 更为重要。

3.1 什么是REST

菲尔丁（http://www.ics.uci.edu/~fielding/pubs/dissertation/web_arch_domain.htm）说："表述性状态转移（REST）架构是为了建模描述现代 Web 如何工作而开发的。"从技术上说，它是与协议无关的，但却是和 HTTP 一起发展起来的。由于这层关系，需要将一些重要的 HTTP 功能谨记于心。

3.1.1 资源

Web 资源是存在于 Web 之上的某种东西，它的定义随着时间不断演变。起先，资源是指拥有静态地址的文档或文件。后来它的定义变得越发抽象，到现在，资源泛指 Web 上一切可识别、可命名、可找到并被处理的实体。比如传统的 HTML 页面、文档、音频文件、图片等。资源还可能指代那些在传统数字图书馆中找不到的东西，比如硬件、人或者一组其他资源的集合。

REST 使用 Web 地址（URI）访问资源，使用动词（HTTP 请求）操作资源。

3.1.2 动词（HTTP请求）

HTTP 1.1 定义了一组动词（或者 HTTP 请求），来表示施予资源的某种行为。有了这些方法，RESTful 的方式就是使用 GET、POST、PUT 和 DELETE。可以这样理解这些动词：就像操作数据库中的数据时使用到的命令。

在数据库里，通常用记录代表数据实体；在 REST 中，对应的实体是资源。HTTP 动词（POST/GET/PUT/DELETE）和在数据库或对象关系管理（ORM）系统中传统的 CRUD（Create/Read/Update/Delete）操作对应。REST 架构规定：访问或修改 Web 上的资源，和

访问或修改数据库中的记录一样，需要使用 SQL 或其他查询的语言。请阅读附录 B 获取 HTTP 1.1 中和 REST 有关的选项清单。

HTTP动词	对资源的操作	对应的数据库操作
POST	新建（或追加）	insert
GET	获取	select
PUT	更新（或新建）	update
DELETE	删除	delete

其他 HTTP 方法

还有其他一些 HTTP 方法也很有用，不过无法和传统的 CRUD 数据库操作对应。HEAD 方法和 GET 相同，区别在于前者的响应不包含响应体。这在利用 HTTP 头的 REST 相关技术中变得很实用。基于浏览器的同源安全策略，跨域资源访问是被禁止的，跨域资源共享（CORS）技术通过 HTTP 头和服务器通信，返回 JSON。有一些方法在 HTTP 头中存放文档链接，时不时会出现重要信息放在响应头而不是响应体中的情况，此时使用 HEAD 调用方法就有意义，而不必像标准的 GET 方法那样花费额外的开销返回响应体。

HTTP OPTIONS 能获取对一个资源可用的 HTTP 请求，这不仅便于调试和支持，而且能帮助创建一个统一接口的、自描述的系统，所有链接都通过一个接入点访问。结合这些代表系统中所有资源访问点的链接，使用 HTTP OPTIONS 实现的系统可以通过一个链接，返回每个资源可用方法的完整列表。

3.1.3　统一资源标识符

在网络系统中，需要某种形式的句柄或地址以访问资源。统一资源标识符（URI）泛指实现这一功能的字符串。URI 可进一步划分为统一资源名（URN，代表资源的名字）和统一资源定位符（URL，代表资源的地址）。

大多数 URL 遵循如下格式：

```
<scheme>://<user>:<password>@<host>:<port>/<path>;<parameters>?<query
key/value pairs>#<fragment identifier>
```

请参考《HTTP 权威指南》[1] 以获取更多关于 URI 和 HTTP 的知识。

在 REST 中，通过元素路径指定资源，URL 中的斜杠界定了资源，并且表示了资源之间的层级关系。REST URL 的示例和更详细的解释详见第 8 章。

在基于 Web 的 API 中，人们使用 URL 定位资源，并且依照一些附加约定表明资源之间

注 1：本书已经由人民邮电出版社出版。

的关系。在 REST 中，这不是必需的，只是出于对 URL 的可读性和一致性所采用的一种风格约定。大部分情况下，URL 应使用小写字母，使用连字符而不是下划线，结尾使用斜杠。单个资源使用单数名词指代；一组资源使用复数名词指代。不鼓励使用文件后缀，但当 API 支持多种格式，比如 XML 和 JSON 时，可使用文件后缀。使用“Accept”和“Content-Type”头来控制格式，是一种更好的替代方案。Mark Masse 所著的 *REST API Design Rulebook*（O'Reilly）一书列举了上面提到的一些约定及其他，但是在实践中，URL 的结构和格式有很多种可能。

使用连字符替代下划线？

需要指出，上文提到的 URL 命名规范不是金科玉律。很多流行的 API，包括 Twitter（https://dev.twitter.com/overview/documentation）和 Dropbox（https://www.dropbox.com/developers/core/docs）都在使用下划线。在过去，有很多因素，比如搜索引擎优化让下划线成为一种不好的实践（在 Google 的搜索引擎索引里，以下划线连接的元素会被串联起来，而使用连字符的字符串则可分解成单词）。关于细节，有很多不同的意见，但人们都明确同意：URL 应该越短越好，易读、清晰，并在 API 间保持一致。

3.2 REST约束

REST 是由一组挑选的约束定义的软件架构风格（http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm）。遵循这些约束设计的系统，体现出了有益的系统属性和良好的工程原则。

3.2.1 客户端–服务器端

客户端 – 服务器端架构直截了当地划分开了职责。它最大化地利用了现代客户端的处理能力——这种能力以往只在高端电脑上具备。服务器端将很多职责转移到客户端，使自身得到了极大简化。这样的系统易于扩展，分成不同层让各层之间可以独立开发。

3.2.2 无状态

REST 要求无状态的设计，会话数据保存在客户端。客户端的每次请求必须“上下文无关”和自包含的。也就是说，请求要包含整个客户端状态。这要求服务器端在不通过额外查询应用状态的情况下，响应所需要的所有数据，并获得与应用状态相关的数据。因为所有的数据都包含在了请求里，使得每次交互的意图理解起来变得容易。因为会话数据不在服务器端维护，容错性和可扩展性都得到提升。服务器资源不会浪费在存储上，否则就需要重新获取。

设计符合 REST 无状态本质的系统有很多挑战。挑战之一是更多和更频繁的请求带来的

额外网络流量（可以通过维护服务器状态的非 RESTful 客户端 – 服务器端应用来保持服务器端会话以减少流量）。所有的请求都包含状态相关数据，那么一定有一些数据被重复发送。服务器不维护会话这一点，意味着客户端有了更多的职责，因此会变得更复杂（这是现代 RESTful 应用中，JavaScript 变得越来越复杂的原因之一）。浏览器存储在这种系统中变得格外有趣。由于 REST 系统通常支持多种客户端，在客户端花些时间和精力计划是必需的。

虽然挑战很多，但无状态带来的好处是巨大的，尤其在部署多服务器时，通过负载均衡分发请求的情况下。如果使用基于服务器端的会话，有两种选择。第一种选择是要求对于指定会话，使用同一个服务器响应所有请求。第二种选择是创建一个可供所有服务器访问的集中式会话存储区，每次作出请求，都需访问这个集中式的数据存储区。该实践（有时称为*会话关联*或*粘连会话*）让扩展变得更难。大规模部署通常要求随时向服务器发布更新，如果使用服务器端会话，必须等待旧会话失效后，才能将接入请求集成进安装更新后的服务器。这是一个复杂、难于操作的过程。如果服务器端不保存状态，就不会有这些麻烦。而且，浏览器的标准功能，比如返回前一个页面或重新加载当前页面，在消除服务器端维护的会话后，都不需要特殊处理即可实现。

REST 的无状态设计对很多开发者来说，是最难理解的特性。它需要对传统 Web 开发实践做出极大调整。由于每个请求都包含状态带来的性能损耗也不是无法克服的，尤其是在考虑到有很多 REST 约束的时候，好处很多。

3.2.3　可缓存

由 REST 的无状态约束带来的性能问题在很大程度上可以通过缓存来弥补。REST 要求给数据打标签，标明其是否可被缓存。这使得客户端应用能重用缓存的响应，而不是此后发送同等的请求。和任何系统的缓存一样，这会极大提高客户端应用性能，但与此同时，为避免数据过期，采用合适的缓存失效策略带来了复杂性。以 Web 来说，缓存可以存在于客户端（浏览器里）、服务器端，或者位于二者之间（网关或代理服务器）。

3.2.4　统一接口

统一接口规定了所有 RESTful 应用的通用资源定位方式。每一个 REST 应用都共享一种通用架构，那些熟悉这种架构的人一眼就能看明白。因此，为特定应用需求定制接口的能力会受到严重限制。不过，该约束也提供了很多灵活性。HTTP 的每个版本都在演进，为 RESTful 设计提供了额外的机制（比如新增的请求类型）。该约束由其他一些限制构成（见表 3-1）。

表3-1：统一接口约束

统一接口约束	描述
标识资源	通过 URI 定位资源
通过表示操纵资源	资源需要表示，它的表示形式就是请求发送的内容（即 XML/JSON 文档）
自描述的消息	使用 HTTP 动词的无状态请求
使用超媒体作为应用状态引擎	HATEOAS：能指明应用状态转变的可用链接

3.2.5 分层

分层设计能让网络模块和每条请求按序交互。每个模块不能越过正在通信的层级访问其他层。这意味着常用的网络设备，比如防火墙、网关能继续使用，可以使用代理服务器进行缓存或转换。

3.2.6 按需交付代码

REST 允许通过 JavaScript 或嵌入浏览器的应用按需交付代码，但这不是必须的。这为扩展应用提供了更多可能性。其缺点是降低了根植于 RESTful 系统中的可见性（特别是像 Java Applet 这样的对象）。

3.3 HTTP响应代码

开发 REST API 时，标准 HTTP 状态码（详见附录 B）为客户端提供反馈、报告潜在错误。响应的本质，直接依赖使用的 HTTP 请求。

什么是成功

关于如何在 Web API 中使用 HTTP 响应代码，业界还未普遍达成一致。这一方面是因为响应没有和 HTTP 动词对应，而是分成 5 种类型：概要信息、成功的请求、重定向、客户端错误、服务器端错误。在某些情况下，响应代码意义明确；另一些时候则依赖具体的 API 调用或 API 设计者的选择。

以成功代码（200）为例，在大多数情况下，这是一个 GET 请求成功后返回的恰当代码。事实上，在 HTTP 1.1（http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html）规范里，它被列为 GET 和 POST 请求的响应代码。响应取决于 HTTP 的请求类型，如表 3-2 所示。

表3-2：HTTP 1.1成功响应代码（200）

HTTP动词	期望的响应内容
GET	和请求资源对应的实体
HEAD	值返回和请求资源对应的 entity-header 属性，不返回消息体
POST	描述或包含动作结果的实体
TRACE	包含服务器端收到请求信息的实体

也有人使用 HTTP 200 作为 PUT 或 DELETE 请求的响应代码，也可能使用其他代码。有些 API 使用 HTTP 201（新建）来响应 PUT 或 POST 请求。204（无内容）可能用于 DELETE 请求，不返回其他响应，如果请求成功但服务器故意不返回内容的话，也可能用于其他动词（甚至是 GET）。如果 Web 应用使用基本认证，可返回 HTTP 401（未认证）注销登录。因此，什么是成功，需要结合上下文来考虑，选择合适的响应代码需要理解代码的含义和请求的性质。

响应代码的第一个数字指明了响应的种类。客户端至少要识别响应的种类，使用响应代码要求一定的灵活性和变化（见表 3-3）。

表3-3：HTTP响应代码的种类

第一个数字	含义
1	信息
2	成功
3	重定向
4	客户端错误
5	服务器错误

Mozilla（https://developer.mozilla.org/en-US/docs/Web/HTTP/Response_codes）、Yahoo Social API（https://developer.yahoo.com/social/rest_api_guide/http-response-codes.html）和 *REST API Design Rulebook*（http://shop.oreilly.com/product/0636920021575.do）一书都周到细致地展示了响应代码的使用。相似性强调了出现的约定和最佳实践，不同点则展示了对于设计给定 API 时，根据其特质所选用的不同设计和区别。

3.4 JSON

道格拉斯·克罗克福德大名鼎鼎，不是因为他强调了 JavaScript 语言的方方面面，而是因为他专注于一个子集。简言之，他对如何使用语言加了自己的限制。他的另一创新是设计了脱胎于 JavaScript 语言、用于数据交换的 JSON（JavaScript Object Notation）。

从现代计算机技术出现伊始，就有各种各样的数据交换格式。除了二进制和一些私有的格式，很多格式希望能兼具人类和机器的可读性。起初，人们使用定长和分隔的文件格式（此格式现在仍然有人在用）。1996 年，人们拟定了 XML 第一版草案，兼顾了人类和机器的可读性，但因其冗长和不必要的复杂性，一直饱受大家诟病。为此，一些具备 XML 某些特性，比如层级结构的语言被发明出来（比如 YAML）。2002 年，道格拉斯·克罗克福德购买了 http://www.json.org 域名，他将 JSON 描述如下：

- 一种数据交换格式（对 XML 的一种轻量级替代品）；
- 一种编程语言模型，是 JavaScript 语言的一个子集；

- 一种易于解析的格式（事实上，如果使用 JavaScript 语言，可对 JSON 字符串调用 `eval()` 方法，就可将其转换为 JavaScript 对象，虽然这样做有点不太安全）。

同时，他强调了 JSON 优于 XML 的地方，如下。

- 相比 XML，响应体更小。在 XML 中，起始和结束标签，以及元数据需要额外的空间。
- JSON 和 Web 的互通性更强。JSON 是 JavaScript 的子集，这让客户端集成变得易如反掌。
- 上述特性带来了性能上的提升，也使 Ajax 调用中的集成工作变得简单。

当然，JSON 也有一些“不值一提”的奇怪行为，如下。

- JSON 不完全是合法的（http://timelessrepo.com/json-isnt-a-javascript-subset）JavaScript。
- JSON 中无法添加注释。
- 如前所述，JSON 是 JavaScript 的子集，因此可使用 JavaScript 的 `eval()` 方法求值。但是，`eval()` 方法是危险的（https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval#Don.27t_use_eval.21）。人们常说“`eval()` is 邪恶的（evil）”。
- 大多数现代浏览器强制使用“同源策略”，这使得来自其他网站的 JSON 无法求值。人们使用 JSONP（或者 JSON-P、带填充的 JSON）从其他域里的服务器上请求数据（通过 HTTP GET），在调用者的环境中，已经定义了一个函数用来维护 JSON 数据，这就是为什么要称作“填充”的原因。最近，人们开发出了 CORS（跨域资源共享，http://www.w3.org/TR/cors/）技术，作为一种更有活力、更安全的技术，有望取代 JSONP。CORS 使用 HTTP 头，让来自指定域的服务器提供资源。同时需要在服务器端做相应配置以生成 JSON。

JSON 中的注释

JSON 里不允许注释这一点有些莫名其妙，因此出现了很多变通方法。方法之一是向 JSON 对象增加一个“comment”元素，然后将注释内容当作元素的值。这里要避免创建两个同名元素，并且默认解析器会选择最后一个。比如：

```
({"a":"This is a comment", "a":"CONTENT"})
```

虽然该片段能被很多 JSON 解析器解析，但这和实现有关，并不是规范中定义的行为。

3.5 HATEOAS

JSON 是一种非常简单和紧凑的格式。和其他为表示数据提供复杂类型系统的格式不同，JSON 只提供了有限的几种数据类型，包括字符串、数字、布尔值、对象、数组和 `null`——对于大多数应用来说，这些数据类型的表达能力和扩展性已经足够强。由于 JSON 作为一种在 RESTful API 中广泛使用的数据交换类型，读者可能会希望有一种类型来表示超链接，但是这样的类型是不存在的！这是有问题的，特别是当 REST 包含一种*超媒体作为应用状*

态引擎（Hypermedia as the Engine of Application State，HATEOAS）的约束时。

该约束限定客户端通过链接（即超媒体）交互。因此，从严格意义上来说，如果 JSON 或者 JavaScript（这是前者的基础）没有链接数据类型，一个 Web API 就不能称为“RESTful”API。

基于 XML 的 REST API 通常使用 Atom 供稿格式（这里链接包含 rel、href、hreflang 和 type）或报文头中的链接。同样的，人们也开发出了包含标准链接的 JSON 扩展：

- HAL（http://stateless.co/hal_specification.html）；
- Siren（https://github.com/kevinswiber/siren）；
- JSON-LD（http://json-ld.org/）；
- JSON Reference（http://tools.ietf.org/html/draft-pbryan-zyp-json-ref-00）；
- Collection+JSON（http://amundsen.com/media-types/collection/format）；
- JSON API（http://jsonapi.org/about/，从 Ember JS REST Adapter 提取而来；Ember JS REST Adapter 参见 http://emberjs.com/）。

W3C 在其官网上有一篇文章“JSON-based Serialization for Linked Data”（http://www.w3.org/TR/json-ld-syntax/，本书写作之时其状态为“W3C Recommendation 16 January 2014”）。

不论表示的实体如何，很多链接都有相似的类型或关系。以集合为例，可以引用第一个、最后一个、下一个或前一个元素。帮助、术语表或者关于链接在很多情境中都是相关的。由于有了这些共性，人们就试图将链接之间的关系（http://www.iana.org/assignments/link-relations/link-relations.xml）标准化（http://tools.ietf.org/html/rfc5988）。建立了关系之后的链接能让自动生成的链接更加具体和可靠。

了解围绕 HATEOAS 的讨论和工作是有价值的，但是最终结果是什么样子，现在还无从知晓。事情远比想象的复杂，每个 API 设计是否要完全遵守这一规则也尚未达成一致。

超链接连接能力的连续性

出现在 Web 上的资源，其连接能力千差万别。一方面，一些格式需要定义严格的链接；一方面，另一些格式却禁止使用链接。

格式	描述	示例
1	自动生成的标准链接	Atom Publishing Protocol（APP）
2	人为指定的标准链接	XHTML
3	扩展形式的链接	链接作为已有格式的扩展（HAL，http://stateless.co/hal_specification.html）
4	字符链接	作为链接的字符串
5	无链接	特殊的交换格式禁止使用链接

菲尔丁定义的 REST 要求合理使用格式 1。很多 Web API 受 REST 启发，有可能选用其他格式。

REST和JSON

随着 REST 被用于除 Web 之外的很多真实项目，逐渐体现出了它的价值所在（正如菲尔丁在论文中表述的那样）。菲尔丁曾经表达过他对那些冠以 RESTful 标签，却不是超链接驱动（http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven）的 API 的失望：

> REST 是着眼于长久的软件设计：每个细节都提倡软件的持久性和独自进化。很多限制直接牺牲了短期效率。遗憾的是，人们擅长短期设计，在长期设计上却表现糟糕。大多数人认为在当前版本发布后，不会再去设计。有很多软件方法将长期设计刻画为错误的、象牙塔里的设计（没错，如果设计不是由真实需求驱动，会变成这样）。

菲尔丁将 REST 定位成一种长期的解决方案值得尊敬。他提出了只要 Web 以及背后的技术保留基础结构就可以长期共存的想法。如此说来，HATEOAS 和 JSON 的关系就没有什么值得调查、展示和深究的了。当尘埃落下，我或许会扩充或改写本书，加入对这些内容的讨论。

总体上来说，菲尔丁的观点非常重要，他是一位创新的思考者，他发明了 REST。但是，现在设计的很多系统并不需要长存于世（或者卖很长时间）。他对 REST 的表述很精彩，值得大家按他所表述的意思去理解。但是，REST 已经被用在了更广阔的环境里，已经成为一种识别微型 Web API 设计的架构风格的标志，很大程度上依赖于底层 HTTP 的功能。

实用的 REST

为什么很多 API 都以这种“实用”的方式来设计？一方面，这是因为 HATEOAS 原则对客户端程序员提出了更高的要求。一个程序员，不管是有意还是无意，如果将 URI 硬编码入应用，会为将来带来很大隐患，服务器端 API 团队可以直截了当地告诉客户端程序员，你没能遵守规范。

虽然 HATEOAS 从理论上说是设计 API 的好方法，但是现实中可能不适用。在设计时，充分考虑到 API 的消费者，以及他们可能使用 API 的方式，以构建你自己的应用是很有必要的。在有些情况下，HATEOAS 并不是正确的选择。

——*APIs: A Strategy Guide*（O’Reilly）

在菲尔丁的论文中，很有意思地提到了 JavaScript 和 REST 的关系（http://www.ics.uci.edu/~fielding/pubs/dissertation/evaluation.htm）。他讨论了 JavaScript 和嵌入式的 Java “applets” 在 Web 上的对决，为何前者笑到了最后。他指出 JavaScript 符合 Web 部署模型，和 HTML 的可见性原则保持一致，包含较少的安全限制，拥有较短的用户可感知的延迟，而且不需要整个单独下载。

传统的 JavaScript 使用方式和 Web 的设计原则一致，它在浏览器里的大量使用，使得 JSON 成为一种吸引人的数据交换选择，它易于产生，易于处理。

遗憾的是，JSON 是 JavaScript 的一个子集，而不是一种超媒体格式，这意味着在 JSON 和严格的 RESTful API 之间还存在龃龉。尽管浏览器运行 JavaScript 和 Web 的设计与 REST 的正式定义保持一致，但使用 JSON 交换数据却并非如此。

没人对使用严格的定义来区分一个 Web API 是不是 RESTful 感兴趣。JSON Web API 是如此流行，REST 设计的影响如此广泛，人们发展出了很多标准来评估 REST。

3.6 API衡量和分类

理查德森在 2008 年（http://www.crummy.com/writing/speaking/2008-QCon/act3.html）提出的理查德森成熟度模型（http://martinfowler.com/articles/richardsonMaturityModel.html）表达了一种连续性：服务可借由 REST 标准来评估。

级别	服务	描述
0 级	HTTP	HTTP 作为一种远程交互的传输系统（远程过程调用）
1 级	资源	不将所有请求发送至单一的服务，而是引用不同的资源
2 级	HTTP 方法	利用 HTTP 动词

Jan Algermissen 基于 REST 约束（详见表 3-4），也提出了类似的分类来描述 API。

表3-4：对基于HTTP的API分类，源自Jan Algermissen（http://nordsc.com/ext/classification_of_http_based_apis.html）

名称	动词	一般媒体类型	特殊媒体类型	HATEOAS
WS-* web services (SOAP)	N	N	N	N
RPC URI-tunneling	Y	N	N	N
HTTP-based Type I	Y	Y	N	N
HTTP-based Type II	Y	Y	Y	N
REST	Y	Y	Y	Y

两种分类系统都保留了菲尔丁对 REST 严格的定义，同时也兼顾了实现 Web API 时必要的妥协。

3.7 函数式编程和REST

函数式编程和 REST 有很多共性。从本质上来说，两者都是声明式的，几乎没有对于控制流程的描述。两者对于控制副作用的观点也一致，维护了引用的透明性，都以无状态的方式操作。拿 JavaScript 来说，如果对函数式编程范式有清晰的理解，会为高效使用 REST

带来极大帮助，也有助于理解 REST 的优点。

> **Web 演算的 Web**
>
> 除了上述对于 REST 和函数式编程从实用性上的比较，Tyler Close 将 Web 视为 lambda 演算的一种变体（http://www.waterken.com/dev/Web/REST/）。由于函数式编程语言基于 lambda 演算，两者之间的关联比看上去更加密切：
>
> > 认识到 Web 是以 HTTP 操作的资源，揭示了 WWW 其实是 lambda 演算的一种变体。资源就是闭包，POST 方法是“apply”操作。由资源组成的 Web 其实就是一个闭包网络。WWW 的创新在于对闭包网络的反思，即 GET 方法……将 HTTP 理解成 lambda 演算的一种变体，让使用 HTTP 作为分布式计算的基础，而不仅仅是分布式超媒体成为可能。

在 *Programming Scala*（http://shop.oreilly.com/product/9780596155964.do，O'Reilly 出版）一书中，作者 Wampler 和 Payne 指出，很多情况下，面向对象系统并不能兑现其组件复用的承诺。相反，成功的组件模型其实很简单。他们对这种相对简单的方式反复重申，在章节末尾对函数式编程做了总结：

> 组件之间应该通过交换不可变的数据结构交互，比如列表和散列表，它们同时携有数据和命令。这种组件模型足够简单，也足够丰富，能应付真正的工作。注意，这看起来多么像 HTTP 和 REST 啊！

3.8 项目

下面的项目创建了一个最简化的 API，能响应 GET、POST、UPDATE 和 DELETE 请求，将处理结果返回给客户端。该项目还展示了测试 REST 和其他 Web API 时用到的各种工具。

创建该项目，需做如下准备。

(1) 从 GitHub 下载项目代码（https://github.com/java-javascript/client-server-web-apps/tree/master/Chapter-3-REST-and-JSON/jruby-sinatra-rest）。

(2) 切换到 jruby-sinatra-rest 目录。

(3) 连接上因特网，使用如下命令构建项目：

```
$ mvn clean install
```

该命令会下载项目所需的 Ruby 和 Java 资源。构建成功后，会看到如下信息：

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 32 seconds
[INFO] Finished at: Tue Apr 30 12:59:15 EDT 2013
[INFO] Final Memory: 13M/1019M
[INFO] ------------------------------------------------------------------------
```

(4) 启动服务器：

```
$ mvn test -Pserver
```

几秒钟后，应用服务器启动成功，准备好接受请求：

```
== Sinatra/1.3.1 has taken the stage on 4579 for development...
[2013-04-30 13:00:24] INFO  WEBrick::HTTPServer#start: pid=29937 port=4579
```

(5) 在浏览器地址栏输入 http://localhost:4579/about。

接下来会得到如下响应：

```
{
  "ruby.platform": "java",
  "ruby.version": "1.8.7",
  "java.millis": 1367342095907
}
```

这表明 JRuby 已经就位，需要的 RubyGems（软件包）也已经就位，Java（系统）可以成功调用。

现在服务器已经运行起来，你可以选择一个客户端来测试 REST 调用。下面的一些例子使用了 Curl（http://curl.haxx.se/）。Curl 是一个命令行工具，通过访问 URL 传输数据。它可以和包括 HTTP 在内的很多协议工作，能处理很多通常在浏览器里完成的任务。

让我们开始吧，可对任意 URL 调用 GET。以我们的服务器为例，返回结果是关于请求的信息：

```
$ curl http://localhost:4579/about
{
          "ruby.platform": "java",
          "ruby.version": "1.8.7",
          "java.millis": 1367342095907
}
```

Curl 有大量选项来改变命令行为和返回结果。在传统浏览器里，一般不显示 HTTP 头信息。使用 `-i`（包含 HTTP 头）或 `-I`（只包含 HTTP 头）在响应中显示 HTTP 头信息：

```
$ curl -iI http://localhost:4579/about
```

服务器端很简单，它由 Ruby 编写（因此运行在 JRuby 之上）。程序用到了一个 Sinatra（http://www.sinatrarb.com/intro.html）的微框架，该框架被形容为"一门使用 Ruby 快速创建 Web 应用的 DSL"。就像所说的那样，它不需要有可能混淆功能的大量额外代码和结构，就可以创建服务器，通过 HTTP 对外提供服务。

前面几行描述了需要引入的包和一些配置。然后，"before"部分定义了 before 过滤器。在所有情况下，内容类型都指定为 JSON，从发来的请求里提取一些值赋给响应，响应代码要么为 200，要么是传入的请求参数 `httpErrorCode` 的值。使用 `-i` 选项查看响应和 HTTP 头：

```
$ curl -i http://localhost:4579/?httpErrorCode=400
```

该例子如在参数中指定的那样，返回 HTTP 400（Bad Request）：

```
HTTP/1.1 400 Bad Request
Content-Type: application/json;charset=utf-8
Content-Length: 197
X-Content-Type-Options: nosniff
Server: WEBrick/1.3.1 (Ruby/1.8.7/2011-07-07)
Date: Tue, 30 Apr 2013 17:43:17 GMT
Connection: Keep-Alive
```

使用 POST 或 PUT JSON 的例子如下所示：

```
$ curl -i -H "Accept: application/json" -X POST -d "['test',1,2]" \
http://localhost:4579

$ curl -i -H "Accept: application/json" -X PUT -d "{phone: 1-800-999-9999}" \
http://localhost:4579
```

下面是 DELETE 的例子：

```
$ curl -i -H "Accept: application/json" -X DELETE http://localhost:4579
```

服务器能处理任意路径（使用通配符匹配），因此可传入任意路径和查询参数，然后在响应中显示：

```
curl -i -H "Accept: application/json" -X POST -d "['test',1,2]" \
http://localhost:4579/customer?filter=current
HTTP/1.1 200 OK
```

还有一些浏览器插件也能用来测试 REST 调用。在这里，我们不介绍这些插件，而是使用应用中自带的、我们自己用 jQuery 编写的一个简化 REST 客户端。前面使用 Curl 描述过的调用，在 http://localhost:4579/testrest.html 输入参数后，如图 3-1 所示。

在 Chrome 浏览器的开发者工具的网络一栏中，可以查看头信息和响应代码。图 3-2 展示了修改上一次调用返回 401 的样子。

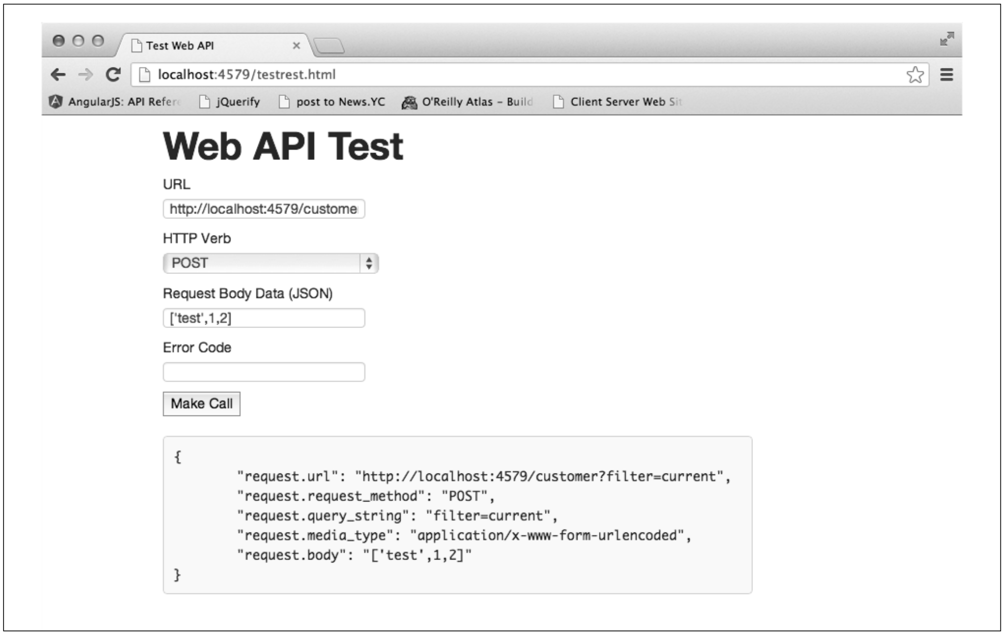


图 3-1：测试 REST

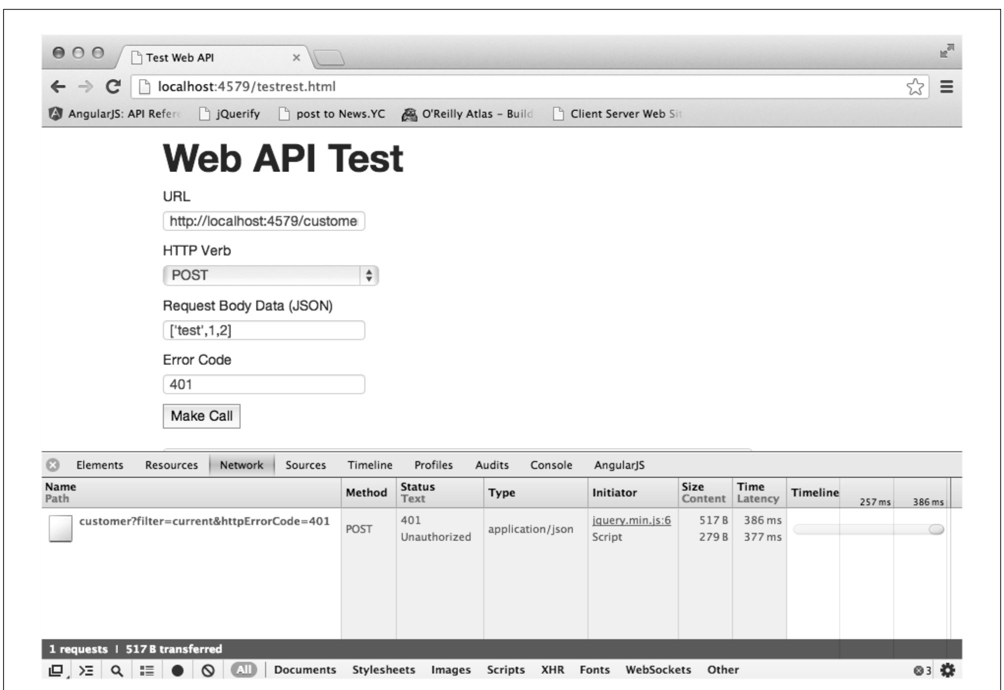


图 3-2：测试 REST 错误

这个简单的应用还介绍了如何通过几行代码，就能搭建起 Web API 作为后台供客户端开发者使用，这样就能和服务器端开发者一起并行工作，实现服务器的完整功能。下面是对这几行代码的简略一览：

```
$ cd jruby-sinatra-rest/src/main
$ find scripts -name *.* | xargs cat | wc -l
141
```

这不包括由内容分发网络（CDN）提供的 jQuery 和 bootstrap，但仍然展示了几行代码带来的无限可能。

3.9 其他Web API工具

Web API 不同任何平台绑定，因此，选用什么样的工具调试更多出于开发者的个人喜好。正如展示的那样，浏览器和浏览器插件非常有用，而 Curl 提供了一种灵活的命令行选择。基于 Java 的 Web 客户端（https://code.google.com/p/rest-client/）、REST 框架（http://restlet.com/）和 Eclipse IDE 插件（http://ww1.ywebb.com/）深受 Java 社区的喜爱，而 Fiddler（http://www.telerik.com/fiddler）则是微软拥趸的人气选择。

3.10 约束回顾

天才们完全抛弃外界影响的想法是不切实际的。这种方式在软件开发领域行不通，与艺术领域里取得的那些伟大成就也没多大关系。相反，认清约束并且创造性地加以应用是一种更为可行的方式。《纽约时报》曾撰文（http://www.nytimes.com/learning/general/onthisday/bday/0617.html）将作曲家伊戈尔·斯特拉文斯基形容为从作曲伊始就明确限制，以“获取执行时的精准”：

> 对斯特拉文斯基来说，作曲就是解决音乐问题的过程：在他开始工作之前，就将这些问题定义出来。
>
> 比如，在谱写“众神领袖阿波罗”时，他写信给负责该芭蕾舞剧的伊丽莎白·斯普拉格·库利奇，询问演出时舞台的大小、大厅有多少座位，甚至管弦乐队面向什么方向。
>
> “限制越多越自由”，他曾经如是言，“限制的任意性，实质上是为获取执行时的精准度服务的。”
>
> ——《纽约时报》

REST 和 JSON 带来的约束远非任意，但是它们的确为客户端 - 服务器端 Web 应用的开发带来了执行上的精度。

第4章 Java工具

> “编程语言最初的作用是充当人类和计算机之间交流的媒介。现在，软件的生命周期延长了，编程团队也壮大了。因为程序员需要讨论软件，代码也就成了人与人之间交流的重要媒介。”
>
> ——吉勒斯·杜博切特

在这段话里，吉勒斯·杜博切特（Gilles Dubochet）在分析编程语言在人类交流中扮演的角色时，介绍了“分布式认知”。人们常常抽象地研究编程语言，却忽略了一些显而易见的环境因素。

- 编程从来不是一项孤立的活动。现在很多项目都需要庞大的分布式团队。
- 计算机语言是程序员与程序员之间交流的重要媒介。
- 程序员用文字描述代码质量，明示或暗示着期望代码在同事之间是可读的。
- 紧凑的代码能在达成共识的程序员之间提高“代码的可理解性”。
- 程序员没办法将大型系统的所有需求都存在自己（人类）的记忆里。
- 人工可读的文档（如果有的话）常常不能和系统保持同步。
- 文档不是总能够描述软件的每一个边界情况。信息隐藏在代码中。

本章提到了各种能运行在 Java 平台上的编程语言，这使为某项特定的任务或特殊人群挑选最佳编程语言成为可能。本章还讲解了 Maven，作为一种工具，它在团队编程时提供了很多最佳实践。

4.1 Java语言

Java 是一种成熟、稳定，并广为人知的语言，但挑战依然存在，在某些地方，这种基于

类、静态类型和面向对象的语言用起来很麻烦。技术可能会彼此不搭，一开始就考虑替代方案可能有助于弥合分歧。

开发者熟悉将一门面向对象的语言（比如 Java）和关系型数据库集成起来的挑战，而这只是很小的一个例子。虽然一些矛盾能通过改变使用习惯化解，但总会引入一个映射层，这是由面向对象和关系型模型所秉持的基本哲学之间的不同造成的。

这并不是 Java 所特有的，JSON 和 REST 之间也存在这种矛盾。REST 要求链接需满足 HATEOAS 中统一接口的约束。JSON 作为 JavaScript 语言的子集，并没有与之对应的结构。不能说谁对谁错，它们的起因不同，集成到一起时就会产生不协调的情况。

同样的矛盾在面向对象和 REST 之间也有。在面向对象系统中，操作的是对象本身。在 REST 中，资源是由它的表示操作的。这层额外的抽象让人们能以一个共同的接口访问资源，这不是一个对象系统中缺省就有的功能。

Java 开发者发明了很多类库来解决大多数翻译和与其他技术集成的问题。此外，Java 虚拟机（JVM）作为 Java 平台的一个组件，本身就为使用其他语言提供了可能。

4.2　Java虚拟机

Java 被翻译成字节码在 JVM 上执行（见图 4-1）。Java 一开始就被设计成跨平台的语言，因此为不同平台，从高性能服务器到嵌入式设备，都开发定制了 JVM。

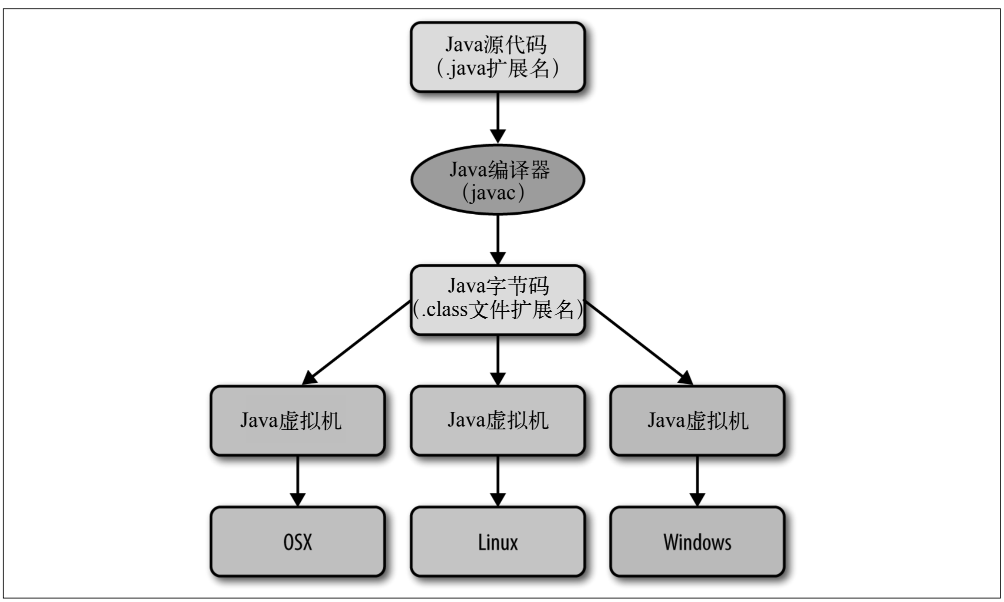


图 4-1：Java 编译流程

JVM 对 Java 语言的相对独立性启示人们开发出了其他运行在 JVM（http://www.oraclejavamagazine-digital.com/javamagazine/20120102?pg=66&search_term=saternos&doc_id=-1&search_term=saternos#pg66）之上的语言，包括 Ruby（前一章介绍的 JRuby）、Python（Jython）、Groovy、Clojure 和 JavaScript。以前人们只把 JVM 当作安装语言的一部分，提供了跨平台开发的能力，现在它变成了实现其他语言的接口（见图 4-2）。

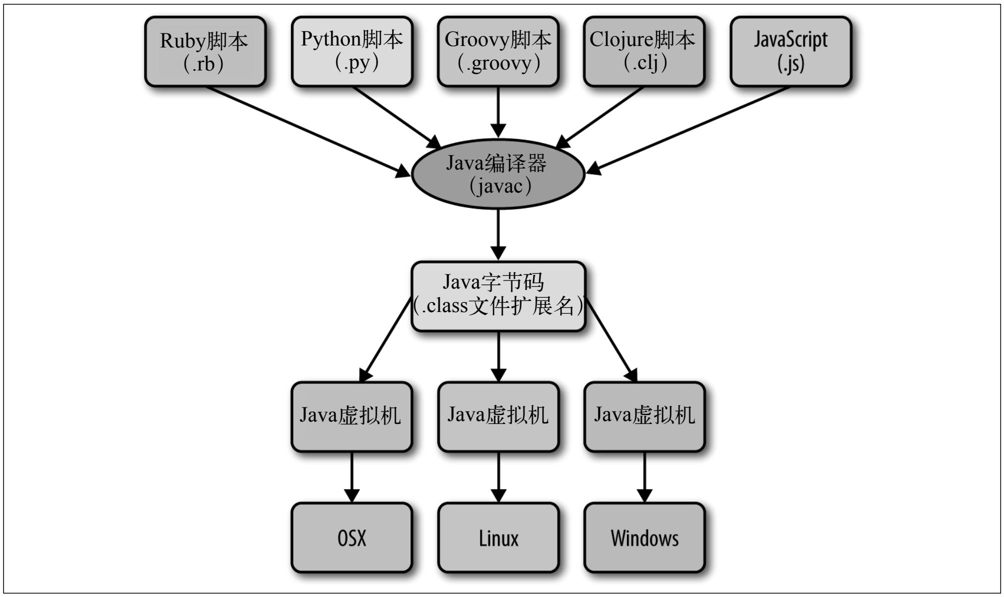
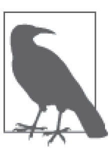


图 4-2：JVM 语言的编译流程



JVM 脚本语言接口和对静态类型语言的支持

图 4-2 所示的语言在本章稍后的项目和书中其他部分都会用到。从众多 JVM 支持的语言中选中它们，是因为知道它们的人相对更多，而且在 JSR 223: Scripting for the Java Platform（https://www.jcp.org/en/jsr/detail?id=223）定义的脚本接口中能使用。脚本接口为在一个项目中集成多种 JVM 支持的语言提供了统一的机制。其中几种语言还受到了 JSR 292: Supporting Dynamically Typed Languages on the Java Platform（https://jcp.org/en/jsr/detail?id=292）后续工作的影响，因此正在加紧开发和改进，以适应 JVM。

有了这些相对简洁和强大的脚本语言，在创建运行在 JVM 之上的 API 时，就可以拿来替代 Java。本章稍后提到的项目将展示如何使用各种 JVM 语言操作 JSON。

4.3　Java工具

Java 是一门成熟的语言，有很多标准的 IDE 可供选择，包括 Eclipse（http://www.eclipse.

org/home/index.php）、NetBeans（https://netbeans.org/）和 IntelliJ（http://www.jetbrains.com/idea/）。这些 IDE 都带有开发工具和插件，把程序员从编写代码、调试、分析和大量其他琐碎乏味的工作中解脱出来。IDE 在开发者工作中占有重要位置，掌握它是现代 Java 开发中一项必不可少的技能。IDE 还提供了和其他系统的集成，比如版本控制和构建自动化，这在团队项目中非常重要。

可用来构建项目的工具有很多。长期以来，Apache Ant（https://ant.apache.org/）都是 Java 社区的最爱。Maven（http://maven.apache.org/index.html）不仅仅定义构建流程，还引入一些广为接受的软件开发任务。Gradle（http://www.gradle.org/）是个基于 Groovy 语言的后起之秀。Ant 和 Maven 都基于 XML，而 Gradle 是一门完整的*领域专用语言*（DSL）。

除去一定程度上的复杂性，还要有额外的架构指导开发流程。项目不同，架构也有所不同，但遵循那些被充分理解和建立的编程实践，就能缩短适应期，在最低限度影响程序员效率的情况下让项目变得可控。简单地说，这种架构能让项目组扩展成为可能。

4.4 构建工具

项目启动时，应该将什么东西包括进来，是架构师或开发负责人要做出的最重要决定之一。这个决定影响到后续开发，当项目稳定后想要改变就很难了。构建系统的选择会对后续活动产生巨大影响，选择时要考虑以下几点。

编程工具语法提供的灵活性非常有用。马丁·福勒（马丁·福勒，http://martinfowler.com/articles/rake.html）曾经对比过 `make`（https://www.gnu.org/software/make，它有为自己定制的语法）、`ant`（http://ant.apache.org/，它的语法基于 XML）和 `rake`（http://rake.rubyforge.org/，它使用了 Ruby 编程语言），以此引起人们对构建脚本应该拥有一门完整功能的编程语言的重视。近年来，另外一个更能引发人们兴趣的因素可能就是构建工具的*依赖管理*。没有依赖管理，开发者就必须确认、定位、下载和安装依赖的模块和相关资源（有时文档还有限）。能自动管理在线代码库里特定版本的模块，对每个开发者来说价值连城，也会让项目更加稳定。请看表 4-1。

表4-1：构建工具的比较

	定制的DSL	XML DSL	编程语言	依赖管理
`make`（https://www.gnu.org/software/make）	Y	N	-	N
`ant`（http://ant.apache.org/ ）	N	Y	-	N
`rake`（http://rake.rubyforge.org/）	N	N	Ruby	N
Maven（http://maven.apache.org/）	N	Y	-	Y
Gradle（http://www.gradle.org/）	N	N	Groovy	Y
SBT（http://www.scala-sbt.org/）	N	N	Scala	Y

这种比较有点过于简单化，因为构建工具都有很好的扩展性。如果构建工具本身不支持，可以将依赖管理做成加载项，也可以让构建工具支持其他编程语言以扩展其能力。比如，使用 Ivy（http://ant.apache.org/ivy/），就能让 Ant 支持依赖管理，还有一个 Groovy 版的 Ant，叫作 Gant（http://gant.codehaus.org/）。

表 4-1 列出了被广泛使用的构建工具，但是并没有列出全部。从 1977 年开始，Make 就或多或少影响了大多数构建工具（包括 Rake）。Rake 又启发了几个面向 Java 的项目，包括 Raven（http://raven.rubyforge.org/）和 Apache buildr（http://en.wikipedia.org/wiki/Apache_Buildr）。

无论项目大小，依赖管理都是毫无疑问的优秀功能。一门完整的编程语言能提供很多灵活性。Ruby 曾经是一个很受欢迎的选择，但现在很多方面已经被 Groovy 超越了，因为 Groovy 太像 Java 了（就像前面提到的，一个合法的 Groovy 文件也是一个合法的 Java 文件）。因此 Gradle 现在开始成为更受欢迎的选择，它包含依赖管理功能，支持一门完整的脚本语言，该语言语法简练，易于被 Java 开发者掌握。类似地，Scala 用户有 SBT（Scala Build Tool，http://www.scala-sbt.org/），它的设计和目的与 Gradle（Gradle 是 Groovy 的构建工具）类似。除了 Scala 中使用 SBT，Play 框架也使用了 SBT，Play 框架同时支持 Scala 和 Java。

尽管选择很多，灵活性常常是有益的，使用一个"自以为是"的工具，基于最佳实践预先做出一些决策能带来很大的益处。Maven 就是这种方式的杰出代表。

需要少一点灵活性吗？

马丁·福勒的文章强调了构建脚本支持一门完整的编程语言带来的灵活性。其优点包括更少的重复代码，没有舍弃 DSL 去做一些"有趣的事"所带来的挫折感。他的焦点在搭建自己的网站上（一个庞大的个人网站）。然而，当更多人参与其中时，更多控制和规范（意味着更少的灵活性）也是有益的。标准的命名规范和熟知的构建顺序会减少项目启动后，陆续加入的开发者的困惑（这在大型的开源项目和按需将程序员分配到不同项目组的公司中很常见）。当蹦出一个要在构建系统中实现一个非同寻常的任务的念头时，系统会让整个事情变慢，做决定的难度更大，以避免做出草率的决定。这就迫使开发团队停下来问自己"我们真的要这样做吗"以及"我们究竟想要达到什么目的"。

一个倾向于灵活性的系统（比如 Gradle）便于任意调用任务（和简单的标准构建相反）。在一个相对较小、较稳定的团队里，交流起来很顺畅，架构很清晰，并且喜欢掌控的感觉，这种系统就能极大提升生产效率。在大一点的团队中，更倾向于标准和规范，因为你无需问"该使用什么命令构建系统？我们有各种各样的任务：compile install、build、make、package……到底该用哪个？"这样的问题，就能检出代码构建项目。Maven 强制程序员遵循标准和规范，这样做节省了时间和交流的开销，让灵活性得以存在于大的团队中。

Maven（http://maven.apache.org/，这个词在犹太语中的意思是“知识渊博的人”）用来组织项目，并且定义软件开发流程。人们称它为“软件项目管理和理解的工具”。它简化了构建流程，提供了创建项目的统一系统（通过声明项目依赖的方式）。它的报告和文档功能负责生产和集中管理所有项目相关的技术产出。

Maven 有一些重要的特性：

- 提倡约定重于配置（同时保留了可扩展性和可定制性）；
- 对所有项目的接口都是一致的；
- 提供了一套定义完好的构建生命周期；
- 基于项目对象模型（pom.xml）定义了通用的项目配置。

想了解跟多关于 Maven 的信息，请阅读 *Maven: The Definitive Guide*（http://shop.oreilly.com/product/9780596517335.do，O'Reilly 出版），本书还可免费在线阅读（http://blog.sonatype.com/2010/01/maven-the-definitive-guide-split-into-two-books/）。

4.4.1 Maven的优点

即使在小型 Java 项目中，Maven 的价值也能立即体现出来，因为 Maven 允许声明式地确定 JAR 文件。作为构建流程的一部分，Maven 在线上仓库里找到这些 JAR，连同它们的依赖一起下载到本地仓库中的合适位置。它负责指定需要的 CLASSPATH，所有初始构建一个项目需要的基本设置在一条命令里就完成了。光凭这条，就足以在构建最小的项目时考虑使用 Maven 了，但是它的优点不止于此。

随着项目的进展和正规化，单元测试、生成的文档和构建报告都会被组装和保存到标准的位置。项目有了标准化的结构和约定，让新的开发人员上手更容易。这对很多项目来说都是大有裨益的，从开源项目到大的组织机构，那里的开发人员会因为人力资源的变化在各项目间转换。

结构良好的项目特别适合开发和部署复杂项目。使用 Maven 构建的项目可以方便地持续集成（CI，http://maven.apache.org/continuous-integration.html）。利用和版本控制系统的关联，使用标签和不断上升的版本号让发布管理变得可控和标准化。事实上，如果系统包含了一组健全的测试，能经常在 CI 服务器运行，并且生产环境里拥有足够的管理和提醒功能，就有可能经常将一些版本持续部署（http://radar.oreilly.com/2009/03/continuous-deployment-5-eas.html）。项目初期形成的合理结构为开发人员带来的优势立竿见影，省去了从开发到部署的很多工作。

4.4.2 Maven的功能

Maven 本身很简单，也没有提供太多功能。通过使用插件，Maven 能执行大量的标准构建

任务，并能被以你能想像得到的任何方式扩展。这里有一些例子仅供参考。

- 使用 JUnit（http://maven.apache.org/surefire/maven-surefire-plugin/examples/junit.html）单元测试，生成单元测试代码覆盖率报告（http://mojo.codehaus.org/cobertura-maven-plugin/），这些都是新建项目时的标准配置（需使用 Maven 项目模板 archetypes）。
- 可以使用 `mvn site`（http://maven.apache.org/guides/mini/guide-site.html）创建一个描述项目目的和状态的项目页面。可以使用类 wiki 格式的 APT（Almost Plain Text）或其他支持格式创建一个 Maven 页面。这个页面统一向外提供了项目信息、联系方式、报告和资源。
- 在开发阶段，使用一行简单的命令：`mvn jetty:run`（http://docs.codehaus.org/display/JETTY/Maven+Jetty+Plugin），就能启动一个最小化配置的嵌入式应用服务器。这个服务器也可以作为构建过程的一部分启动，来支持单元测试。还有其他的插件，可以将 Web 应用部署到你选择的应用服务器上。
- 虽然源自 Java 社区，人们还编写了一些 Maven 插件支持使用其他语言开发。JavaScript 插件（http://mojo.codehaus.org/javascript-maven-tools/javascript-maven-plugin/）提供了代码组织服务和单元测试。还有提供源码压缩（http://alchim.sourceforge.net/yuicompressor-maven-plugin/index.html）和控制代码质量（http://mojo.codehaus.org/jslint-maven-plugin/）的 Maven 插件。

人们常常形容 Maven 是一款“有主见的软件”。从负面理解，这意味着 Maven 很难适应一些特定类型的项目。但是它这种武断的特质却定义了一个即使使用其他构建系统，也值得考虑的良好的软件开发生命周期和最佳实践。它消除了依赖个人记忆和手工定义合适的构建流程和细节，使用标准配置的项目管理可通过大量插件来增强。这在项目，甚至一个组织内提倡了一种统一的实践。和开放的系统将职责推给开发者去记忆相比，Maven 的“主见”让一般开发者根本不用去想这些外围问题，只需要做正确的事就够了，毕竟那些错误的选项会花费额外的精力。

在一个项目中，起初的选择对后续开发的影响极大，不仅对组成项目的组件如此，对开发工具和构建系统同样如此。选择 Maven 还是其他构建工具，这对后续活动影响颇大。本章后面的项目使用了 Maven，这是为了展示创建一个包含最小配置的项目是多么简单，同时也展示了声明式管理模块的价值。一旦项目使用 Maven 构建，集成新的插件和功能就变得非常简单了。

4.4.3　版本控制

如果你已经在阅读本书，那么就不需我赘言使用版本控制的好处。大多数开发机构都需要维护控制他们的软件资产，版本控制系统就是用于这个目的的。如果仅把它当作一个文件系统的备份，就会错失很多伟大的功能。这些功能包括查看代码的变更历史、对比不同版本、标记待发布的代码（标签）、创建分支以便并行开发代码和创建缓存的。

版本控制系统（Version Control System）是开发大型项目时，多组开发人员共同高效合作的基础。当多名开发人员需要修改同一个文件时，极大地降低了解决冲突的难度。使用VCS 支持遗留项目非常简单，因为它保存了历史记录，谁改变了哪些文件，什么时候改的都一目了然。如果提交代码时加上了有用的注释，或者已经和缺陷管理系统进行集成，就能知道为什么会有这样的改动了。

个人开发者能从 VCS 中受益，因为他们有信心去实验，反正在需要时可以恢复到上一个“好”版本。如果一个项目需要持续很长时间，VCS 历史可以展示进度，提供代码变动的原因。

Maven 项目（http://maven.apache.org/source-repository.html）使用 Git 和 Subversion 来管理自己的代码。

4.4.4 单元测试

很容易使用 Maven/Junit（针对 Java）、node/Karma（针对 JavaScript）或其他语言使用的测试框架将单元测试纳入项目之中。还有 Java 框架支持其他类型的测试。可使用 JBehave（http://jbehave.org/）做行为驱动开发（Behavior-Driven Development，BDD），它提供了Maven 插件（http://jbehave.org/reference/stable/maven-goals.html）可供选择。还可以使用 Selenium 插件（http://www.seleniumhq.org/）在浏览器上运行自动化测试（http://mojo.codehaus.org/selenium-maven-plugin/）。

单元测试遇到的挑战之一是将代码和外部依赖分离。在测试时，可以创建一个对象代替真实的对象。有很多这样的对象可供使用。马丁·福勒（http://martinfowler.com/articles/mocksArentStubs.html#TheDifferenceBetweenMocksAndStubs）强调了虚设对象、虚拟对象、桩和模拟对象。测试中使用多少模拟对象因人而异，但 Mockito（https://code.google.com/p/mockito）和 JMock（http://jmock.org/）这样的项目的存在，让原本很难或无法编写的测试代码变得很容易。

项目、客户、开发团队和可用资源不同，适用的测试方式也不同。无论做何选择，Maven都让包含一个测试框架（http://maven.apache.org/surefire/maven-surefire-plugin/examples/inclusion-exclusion.html）变得容易。正如前面指出的，一套全面的测试让很多开发和部署方式成为可能，如果没有这样的验证和测试覆盖，这就是完全不可行的。

4.5 处理JSON的Java类库

有很多 Java 类库能处理（解析或生成）JSON。由于 Java 的基本结构是 Java 对象，使用Java 编写的 JSON 类库基于 JSON 和 Java 对象之间的映射，提供了从对象到 JSON 的序列化和反序列化的方法。Jackson（http://jackson.codehaus.org/）和 Gson（https://code.google.com/p/google-gson）是两种常用的 Java 类库，可以在 Java 对象和 JSON 之间做转换。

4.6 项目

这些项目提供了最小化的项目对象模型，以展示声明式管理模块的价值。一旦项目能成功在 Maven 上构建，增加新的插件和功能就变得很简单。

每个项目均可在根目录下使用下述命令构建：

```
mvn clean install
```

电脑必须联网，这样 Maven 才能定位和下载每个项目需要的模块。第一次在项目中运行该命令时，它会先检查本地仓库（默认在“< 操作系统的用户主目录 >/.m2/repository”下）。如果找不到，会查找 Maven 的在线仓库并下载至本地，以便后续使用。运行 `mvn clean` 命令会移除项目下的资源，但是不会影响到下载至本地仓库的模块。

4.6.1 用Java处理JSON

java_json（https://github.com/java-javascript/client-server-web-apps/tree/master/Chapter-4-Java-and-Tools/java_json）项目展示了 Maven、Jackson 和 JSON Java APIs 的基本用法。pom.xml 需要包含 JSON 和 Jackson，如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>JSON_Java</groupId>
    <artifactId>JSON_Java</artifactId>
    <packaging>jar</packaging>
    <version>1.0</version>

    <dependencies>
        <dependency>
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>2.2.3</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>org.codehaus.jackson</groupId>
            <artifactId>jackson-mapper-asl</artifactId>
            <version>1.9.12</version>
        </dependency>
    </dependencies>
    <build>
      <plugins>
              <plugin>
                <groupId>org.codehaus.mojo</groupId>
```

```
                <artifactId>exec-maven-plugin</artifactId>
                <version>1.2.1</version>
            </plugin>
        </plugins>
    </build>
</project>
```

该项目对象模型定义了该项目的身份（或者坐标：groupId、artifactId 和 version）和打包方式（JAR）。通过 version 指定了 Jackson 和 JSON 的版本，还有一个插件说明了主函数的执行方式。这是一个非常简单的 pom.xml，展示了用 Maven 管理 JAR 之间的依赖有多简单。即使你是一个 Maven 新手，也不会对此望而生畏，而那些 Maven 专家则可以在此基础之上增加单元测试、文档、报告和所有喜欢的花哨功能。

Java 代码由 3 个类组成。定义了一个普通 Java 对象（POJO）来序列化和反序列化 JSON：

```
package com.saternos.json;

public class MyPojo {
  private String thing1;
  private String thing2;

  public MyPojo(){
    System.out.println("*** Constructor MyPojo() called");
  }
  public String getThing1() {
    return thing1;
  }

  public void setThing1(String thing1) {
    this.thing1 = thing1;
  }

  public String getThing2() {
    return thing2;
  }

  public void setThing2(String thing2) {
    this.thing2 = thing2;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    MyPojo myPojo = (MyPojo) o;

    if (thing1 != null ? !thing1.equals(myPojo.thing1) : myPojo.thing1 != null)
        return false;
    if (thing2 != null ? !thing2.equals(myPojo.thing2) : myPojo.thing2 != null)
        return false;
```

```
        return true;
      }

      @Override
      public int hashCode() {
        int result = thing1 != null ? thing1.hashCode() : 0;
        result = 31 * result + (thing2 != null ? thing2.hashCode() : 0);
        return result;
      }
    }
```

DemoJSON 类初始化了一个 POJO，导出它的 JSON 形式，然后又解析 JSON 重新生成一个 POJO，代码如下：

```
MyPojo pojo = new MyPojo();
//……生成pojo对象的代码请参考项目源码

Gson gson = new Gson();
String json = gson.toJson(pojo);

MyPojo pojo2 = gson.fromJson(json, MyPojo.class);
```

DemoJackson 类的逻辑类似，不过使用了 Jackson API：

```
MyPojo pojo = new MyPojo();
//……生成pojo对象的代码请参考项目源码

ObjectMapper mapper = new ObjectMapper();
String json = mapper.writeValueAsString(pojo);

MyPojo pojo2 = mapper.readValue(json, MyPojo.class);
```

可使用如下命令运行 Java 类：

```
$ mvn exec:java -Dexec.mainClass=com.saternos.json.DemoJackson

$ mvn exec:java -Dexec.mainClass=com.saternos.json.DemoGSON
```

4.6.2 用JVM上的脚本语言处理JSON

前面这个 Java 的例子将 JSON 转换为 Java 对象，在其他语言中应该怎么办呢？ JSON 基于一些简单的数据结构（JavaScript 的数组和对象）。在其他语言中这对应数组（或列表）、散列（或字典、表、散列表）。很多类库提供了在 JSON 对象和相应的原生数据结构之间互相翻译的功能。

jvm_json（https://github.com/java-javascript/client-server-web-apps/tree/master/Chapter-4-Java-and-Tools/jvm_json）项目展示了如何使用 Clojure、JavaScript、Jython 和 Groovy 读取一个 JSON 文件，并且获取和输出一个成员的值。图 4-3 展示了如何使用 ScriptEngineManager 类，根据文件后缀调用正确的引擎。

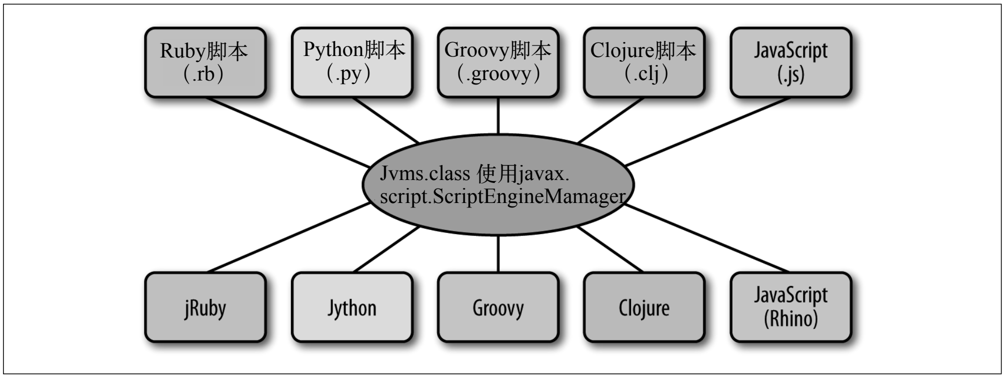


图 4-3：管理脚本引擎



基于 JVM 的 JavaScript 引擎

本书的示例使用 Mozilla 开发的 Rhino JavaScript 引擎。JavaScript 引擎的名字因 O'Reilly 出版的一本书（http://shop.oreilly.com/product/9780596101992.do）而得名。Oracle 公司正在开发一个新的、叫作 Nashorn（http://openjdk.java.net/projects/nashorn/）的 JavaScript 引擎，将会随 Java 8 一起发布。

该项目的 pom.xml 包括了对使用编程语言的相关依赖，而且还配置了如下两个插件。

(1) Exec Maven 插件（http://mojo.codehaus.org/exec-maven-plugin/）让 Java 类和其他可执行程序能在命令行里执行。

(2) Maven Shade 插件（https://maven.apache.org/plugins/maven-shade-plugin/）是众多能打包 JAR 文件的插件中的一种。它与众不同的一点是能将那些命名冲突的项目打包成 JAR。当要实现前面提到的脚本接口时，这种情况司空见惯。

该项目对象模型还包含了对含有引用模块的仓库的引用。

Java 主类（`Jvms.java`）通过参数读入一个脚本文件，然后使用对应的脚本引擎执行（基于文件的后缀名）。

`Jvms.java` 有一个简单的 `main` 方法，用迭代处理通过命令行传入参数。对于每一个参数，程序先尝试验证是否存在这样一个文件，如果没有找到，在文件名前加上 src/main/resources/scripts 作为文件的路径。这个目录就是存放脚本文件的地方。为了和当前讨论的内容相关，已省去异常处理和其他无关细节。如果文件没有找到，则直接从 `main` 方法中抛出异常（它的定义里就有 `throw Exception`）。文件名里第一次发现点的位置以后的文字就是后缀名。最后初始化 `ScriptEngineManager`，根据后缀名得到一个相应的脚本引擎。从文件系统里读取文件，根据后缀名交给对应的引擎求值：

```
public static void main(String[] args) throws Exception
{
  for (String fileName : args) {

    if (!fileName.trim().equals(""))
    {
      File file = new java.io.File(fileName);

      if (!file.exists())
        fileName = "src/main/resources/scripts/"+fileName;

      String ext = fileName.substring(
        fileName.indexOf(".") + 1,
        fileName.length()
      );

      new ScriptEngineManager().getEngineByExtension(ext).eval(
        new java.io.FileReader(fileName)
      );
    }
  }
}
```

可通过 Maven 执行该程序，一次可传入一个或多个脚本作为参数：

```
mvn -q \
exec:java -Dexec.args="testJson.clj testJson.js testJson.groovy testJson.py"
```

如果你喜欢，也可以不使用 Maven，而是用附带的脚本直接执行 JAR 文件：

```
jvms.sh testJson.clj testJson.js testJson.py testJson.groovy
```

Python（Jython，参见 http://www.jython.org/）的例子简单易懂。块内使用对齐，同时使用 `begin` 和 `end` 标致代码的开始和结束，没有多余的让人分心的东西，以一种近乎伪代码的形式实现程序功能。Python 是一种非常规整的语言，由于其设计的一致性，代码看起来不易引起歧义：

```
import json

print('*** JSON Jython ***')

for item in json.loads(open("data/test.json").read()):
    print item['title']
```

首先从文件系统中导入处理 JSON 的类库，然后在对象上迭代，打印出每一个标题。Groovy 的例子遵循近似的模式。Groovy（http://groovy.codehaus.org/）深度基于 Java（一个合法的 Java 文件通常也是一个合法的 Groovy 文件），它引入了一些比纯 Java 更紧凑简洁的用法。这让它成为一个受人欢迎的、向 Java 开发者推荐的语言，他们既能沿用已有的 Java 知识，也能学习 Groovy 的特色：

```
import groovy.json.JsonSlurper

println "*** JSON Groovy ***"

def json = new File("data/test.json").text

new JsonSlurper().parseText(json).each { println it.title }
```

Clojure 的例子也遵循相似的形式，不过你也许会注意到大量的括号。Clojure（http://clojure.org/）是 LISP 的方言，它和其他例子中类 C 语言的语法形成了强烈的对比：

```
(require '[clojure.data.json :as json])

(println "*** JSON Clojure ***")

(def recs (json/read-str (slurp "data/test.json")))

(doseq [x recs] (println (x "title")))
```

JavaScript 版本的例子略有不同。没必要导入任何东西（我们使用 `eval()` 方法仅作展示之用）。如果你还记得，JavaScript 没有内置的文件读写功能。因此这个例子用了点技巧，调用一个 Java 静态方法读取文件，然后将文件内容（Java 字符串）转换成一个 JavaScript 字符串：

```
println('*** JSON Javascript ***')

// 调用一个Java静态方法读取文件,并将文件内容转换成一个JavaScript字符串
var str = String(com.saternos.app.Jvms.readFile("data/test.json"));

var o = eval(str);

for (var i=0; i < o.length; i++){
    println(o[i].title);
}
```

尽管这种实现方式并不纯正，但它展示了在 JVM 之上集成编程语言是多么的方便。

4.7 小结

人们学习一门新的编程语言，或像 Maven 这样的辅助工具的原因很多。最常见的原因之一是因为项目需要。很多人都是在开发 Rails 应用或者使用 Chef 或 Puppet 做系统管理时才开始使用 Ruby 语言的。科学家则因为在工作中需要相关的类库，而被设计规范、性能优异的 Python 语言所吸引。

研究表明自然语言影响思考方式。《华尔街日报》上一篇文章（http://online.wsj.com/news/articles/SB10001424052748703467304575383131592767868?mod=WSJ_LifeStyle_Lifestyle_5&mg=reno64-wsj&url=http%3A%2F%2Fonline.wsj.com%2Farticle%2FSB1000142

4052748703467304575383131592767868.html%3Fmod%3DWSJ_LifeStyle_Lifestyle_5）论述了该观点。文章描述了语言是如何深刻影响人们看待和思考周围世界的方式：

> 关于语言影响思考方式的一些发现如下。
>
> - 俄语中有更多关于浅蓝和深蓝的词汇，因此说俄语的人能更好地区分蓝色系。
> - 一些原始部落的人不说左右，只说东南西北，因此他们的方向感更强。
> - 毗拉哈人的语言中没有数字，而是喜欢使用很少或很多这样的词，因此他们无法精确计数。
> - 一项研究表明，说西班牙语和日语的人无法记起偶发事件中的人物，这点比不上说英语的人。为什么会这样？在西班牙语和日语中，肇事者被省略了：他们说“花瓶碎了”，而不说“约翰打碎了花瓶”。

在特定领域表达能力越强的语言越能让人在那个领域里如鱼得水，编程语言也是如此。这样看来，目的不是学习一门新的编程语言，也不是只为了完成某个项目，它帮助你用不同的眼光来看世界，从整体上提高了你解决问题的能力。大多数学习 Clojure（或其他 LISP 方言）的人并不是出于项目需要，而是为了提高自己思考和解决问题的能力。在现存语言（或所有可能的语言）中，LISP 方言以其简单、表达能力强、强大和灵活著称。对于学习其他语言也是如此，只不过程度上可能不如学习 LISP 方言这么深。每种语言都有自己的特性和社区，和其他语言相去甚远。但是很多差别并不是绝对的，即使不会马上用到，程序员也能通过学习其他语言和工具帮助自己成长。

本章从和其他开发者共同开发的角度展示了几种基于 JVM 的语言和 Maven。某些语言可能利于开发者之间的交流，它们能以更好的方式封装需求，方便日后对项目进行支持和逆向工程。Maven 能用来组织项目资源和开发流程，在很多团队和开发者中，都被证明能促进项目的成功。尽管读书是一种个人行为，但程序员的很大一部分工作都是和其他程序员合作完成的。基于 JVM 的语言和 Maven 为项目提供了合适的功能，可以帮助开发者在漫长的开发周期中和其他开发者交互。

第 5 章

客户端框架



“如你所知，任何事情的开始阶段都是最重要的……因为这段时间其特征正在形成，预期印象也更容易被接受。”

——柏拉图

5.1 概述

在 Web 面世之初，创建一个新的 Web 页面需要先打开文本编辑器，然后从头开始创建 HTML 文档。现在仍然可以这样创建用于教学的小示例或者测试分离出来的 JavaScript 功能片段，然而大部分现代 Web 应用并不是这样开始的，而是使用一个切实可行的项目模版，包含组织良好的目录结构、一些 JavaScript 库、CSS 文件、HTML 和其他资源文件的组合。不同项目间的选择可能不同，但一般都会处理项目一致性、跨浏览器兼容性、设计合理性、软件开发实践（如单元测试），以及性能的优越性等问题。

一直以来，起始项目都是在 IDE 中创建新项目时生成（通过像 Maven 这样的工具，利用原型）或者指定的。这些项目往往与生成工具绑定。Web 开发没有标准 IDE 和构建工具，起始项目没有这样的绑定（虽然有到工具和 IDE 的集成）。起始项目的复杂度和目标不同，但是成功的起始项目都有共同特点：理解、配置和部署都非常简单。它们提供了基本的基础结构来减少和 Web 应用主用途没有直接关系的枯燥的手工活。

起点的选择包括了构建客户端 Web 页面的基础模块：运行在 Web 浏览器中一个 Web 页面环境里的 HTML、CSS 和 JavaScript。

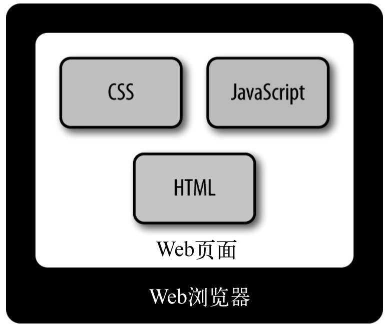


图 5-1：客户端 Web 页面

考虑最多的环境就是客户端自身。浏览器有很多种，每种浏览器都有多个版本。浏览器不再局限于桌面应用，还能在移动设备上运行。可运行每种浏览器实例的特定硬件其功能可能有极大的差别。硬件的处理能力、磁盘空间、内存、屏幕尺寸以及显示特征都可能不同。也许这很明显，但常常会被一些开发者忽略，他们往往只关注在开发机上运行的几款浏览器。

起始项目可以面向特定的内容、样式以及行为。HTML 定义了页面的基础结构和内容；CSS 定义了样式和展现形式；JavaScript 提供了行为能力。设计师可能会选择更小或 CSS 样式更灵活，但是支持各种插件、不需要什么编程的 JavaScript 库的起始项目。而开发者可能会选择设计还过得去、能立即使用，但是利用了最新浏览器功能的前沿 JavaScript 库的起始项目。

起始项目的选择也是基于应用需求和受众的。有的起始项目可能以最大范围的浏览器兼容性为目标，而有的起始项目则以特定移动设备的高度优化应用为目标。例如，像 PhoneGap（http://phonegap.com/）框架包含了特别针对移动设备的起始项目。如果是一个游戏或其他需要大量图形，例如模拟的项目，它的起始项目（https://www.npmjs.org/package/generator-game）则会包含与图形和物理引擎相关的 JavaScript 库。

特定的需求可能意味着它的起点不那么明显或者不那么流行。我们很容易迷上某种框架或设计趋势，但是，项目的特性可能需要不同的方法。例如，如果要主动针对某个浏览器（因为浏览器的内部标准、目标用户群等），可能需要针对浏览器的怪癖进行开发，并选择一个覆盖了这些问题的起始项目。此外，除了标准的 Web 页面以外，HTML、CSS 和 JavaScript 还可以用于开发各种应用，例如浏览器扩展插件和原生应用。针对各种浏览器和设备的通用方法通常是最优的，但一定不是唯一有效的。

现如今，大部分正规的 Web 项目通常都希望开发出来的应用能够跑在各种设备上并且所有功能都能正常运行。绝大多数项目创建时都有这样的愿景，那就是项目能够发展得相对大而全。这意味着要用到与标准软件模式及代码组织相关的客户端 JavaScript 框架。显示、

浏览器功能和设备功能的区别，要求我们使用考虑周全的设计去适配各种设备，并在功能不可用时优雅降级。在应用的生命周期中用到的设备特性会要求使用额外的库和框架，而传统的、仅限于桌面的浏览器应用则无需考虑。

> **项目初期决策的高成本（或价值）**
>
> 看上去本节可能在重复一些显而易见的事情。那就选择一个起始项目开始做吧！
>
> 需要在初期做出深思熟虑的选择，因为这个关键问题会对后面的过程产生巨大的影响。经济学家和社会科学家使用了术语路径依赖（path dependence，https://en.wikipedia.org/wiki/Path_dependence）来描述这种概念，其基本思想是“初期决策是后来情况的歧化起因”。项目初期引入的问题很可能到后来无法修复。例如，选择一个不成熟的 JavaScript 库可能需要大量的补丁和技巧来处理浏览器兼容问题。虽然在开发初期不明显（此时开发者关注的是现代浏览器），但后期的支持会很繁重，而且大量的补充代码可能会让事情变得无法挽回。相反，一个考虑周全的起始项目则可以减少项目实现中繁重的支持任务。

5.2　起点一：响应式Web设计

2010 年 5 月 25 日，Ethan Marcotte 题为“Responsive Web Design”（即“响应式 Web 设计”，英文简写为 RWD，参见 http://alistapart.com/article/responsive-web-design）的文章出现在了 *A List Apart*（http://alistapart.com/）上。响应式 Web 设计现在已经成为概括他所表达的这种观点的通用术语。这种设计风格不对每种设备显示做特定设计，而是努力提供一种在各种设备上都最佳的视觉体验。和软件的设计模式（http://oreil.ly/HF-Design-Patterns，灵感来自于 Christopher Alexander 的架构模式）一样，这个概念来自于建筑架构。响应式建筑会考虑物理空间在有人经过时会如何响应。而响应式 Web 设计则力求基于设备的功能，来适应各种显示设备上不同交互方式的用户体验。RWD 有三大组件，如下。

- 流体网格（Fluid grid，http://alistapart.com/article/fluidgrids）
  用于网页上排版网格的适配。这种技术利用了 CSS 的相对大小来保证网格及其内容在上下文里的显示保持合适的比例。

- 弹性图片（Flexible image，http://unstoppablerobotninja.com/entry/fluid-images）
  包括使用 CSS 的 `max-width` 属性来让图片和其他媒体在其父元素里渲染，以及其他相关的用于避免定点样式的技巧，大小固定的样式会导致无法适配所有显示设备。

- CSS3媒体查询（CSS3 media query）
  根据页面渲染环境的物理特征使用特定的显示单位。

流体布局、弹性图片（依赖于相对大小）以及 CSS3 媒体查询（针对固定大小的响应式布局）组成了 RWD 的基础。流体布局能等比例缩放，而自适应布局会在给定点改变布局。这样可以避免只用流式设计可能引起的失真，以及只做适配设计时中间显示状态的缺失。

使用 RWD 的这三种方式，仅需要一套构建良好的 Web 资源就可以创建在各种设备上查看和有效使用的页面。对于这个问题，Marcotte 后来在他的一本书（http://www.abookapart.com/products/responsive-web-design）中对这篇文章介绍的概念做了扩展。许多项目都是基于这些原则开发的，HTML Boilerplate 和 Twitter 的 Bootstrap 是其中的佼佼者。

5.2.1　HTML5 Boilerplate

HTML5 Boilerplate（http://html5boilerplate.com/）提供了一套满足 RWD 要求的资源。它的特色是包括标准的目录结构、与网站和 Web 服务器配置相关的模板文件。它还包括集成了 `normalize.css`（https://necolas.github.io/normalize.css/，提供了一致的符合现代标准的、跨浏览器的渲染样式）的 CSS，额外的默认 CSS 样式、通用帮助函数、媒体查询占位符，以及打印样式。著名的 jQuery 和 Modernizr JavaScript 库也包含在内。

5.2.2　Bootstrap

Twitter Bootstrap（http://getbootstrap.com/2.3.2/），被它的创建者称为“用于快速开发 Web 应用的前端工具箱”（https://dev.twitter.com/blog/bootstrap-twitter）。这是一个对 CSS 样式和 HTML 结构有特别约定的集合。它应用了一些最新的浏览器技术，以超轻量的资源体积（gzip 格式仅 6 KB）提供了美观的排版、表单、按钮、表格、网格、跳转以及其他所有功能。

就此看来，它比 HTML5 Boilerplate 更优越。它的功能在第 2 版（https://dev.twitter.com/blog/say-hello-to-bootstrap-2）得到了进一步增强，而且在社区中得到的反馈信息也帮助了它不断地完善（http://blog.getbootstrap.com/）。使用 Bootstrap 构建的站点饱受批评，人们认为它们过于单调、没有惊喜，基本上都雷同。使用标准的资源和样式势必会造成雷同，但实际上通过手动定制或者使用主题还是有很大的灵活性的。随着时间推移，这个框架也变得更加“组件化”，某些功能可以添加或删除，同时还有一些生成工具可以定制不同的初始资源集。

网上还有许多类似的项目。如 Zurb Foundation（http://foundation.zurb.com/），它的用户基数较小，而且声称主要针对移动设备。如果你对更轻量的，集成了最小响应式设计且带有一些基本样式的基础网格系统感兴趣，那么 Skeleton（http://www.getskeleton.com/）可能值得看看。如果你的项目仅仅需要专业的、均衡的响应式站点，目前的最佳选项还是 Bootstrap（可通过使用主题和一些手动调整来增强）。

5.3 起点二：JavaScript库和框架

语言不断发展成熟的过程中会涌现各种标准库。在 JavaScript 中，库不需要任何特殊结构或者打包，它们只是一些 JavaScript 文件。目前已有各种创建好的库可用来帮你完成你想做的几乎任何事情，但同时也有大量适用于客户端 Web 应用的通用库。

5.3.1 浏览器兼容性

虽然 JavaScript 是事实标准，但某些浏览器却特立独行。浏览器生产厂商不太可能创建完全兼容的实现。浏览器厂商要在两个方面花费很大的精力，一方面是让自己和别的浏览器不一样，另一方面，在功能的实现上是为了驱动标准而不是遵循已有的标准或指南。幸运的是已经有不少库对这些粗糙的边界做了平滑处理，从而能够写出结果一致（以及更少错误或者破坏性的结果）的 JavaScript 代码，而不需关心浏览器的具体版本。表 5-1 展示了这些库。

表5-1：浏览器兼容性

库	用途
jQuery（http://jquery.com/）	DOM 遍历和操作
Modernizr（http://modernizr.com/）	浏览器特征检测
Underscore（http://underscorejs.org/）	包括了对象和数组操作的工具函数

jQuery 和 Modernizr 是各自领域的事实标准，Underscore 地位的稳固性稍差。另外还有一些库提供了类似的功能：简洁一致、适用性广泛的对象和数组处理方式（例如 Lo-Dash，参见 http://lodash.com/，其愿景就是成为优于 underscore 的替代品）。

5.3.2 框架

小到中型 JavaScript 项目直接进行 DOM 操作（比如用 jQuery）就好了。对大项目而言，提供可交互性数据以及诸如数据校验等附加功能的 JavaScript 类，那么事情将会更简单。这些类可以被填充并与页面上的图形化元素相连接。这种设计的好处在于避免了直接的 DOM 操作。取而代之的是，容器对象中数据状态的变化会反映到页面上。与页面的交互会引起模型状态的修改，模型状态的改变会扩散到所有受影响的视图组件。许多年前第一个 MVC 框架就是为这种用途创建的，现代 JavaScript 框架都已采用了这种模式。MVC 有几种变体，其中包括模型 – 视图、表现器（Model-View Presenter，MVP）和模型视图、视图模型（Model-View View Model，MVVM），因此术语 MV* 有时候被作为统称，用于将这些模型统一标记为一个组。

在外部看来选择初始框架会很困难。如果没有特别的偏好，可能需要花点功夫来决定到底使用哪个框架。除了自身的开发技巧以外，有一些基本的选择标准：框架提供的功能及其流行程度。

5.3.3　功能

MV* 框架非常之多，且其组成的这一列表不断变化。TODO MVC（http://todomvc.com/）这个站点提供了流行框架之间直接而详细的比较。TODO MVC 利用不同的 MV* 框架实现了全功能的 TODO 列表应用，这个站点允许你来比较这些不同的实现。

5.3.4　流行程度

最流行的未必就是最好的，但框架的流行程度是选择它的合理理由，流行程度代表了框架是否有可靠的学习、改进以及 bug 修复的生态系统。Google Trends（http://www.google.com/trends）展示了某些搜索词的当前搜索数，StackOverflow 标签（https://stackoverflow.com/tags）的数量也可以让人了解开发者对一个主题的讨论热度。要更好地理解实际使用了什么代码，可以查看 GitHub 仓库（https://github.com/trending，以及还没有登上列表头条的新项目的数据统计），或者使用像 BuiltWith（http://builtwith.com/）这样的站点的数据来了解总的部署情况。

如果你决定要用某种主流的 JavaScript MV* 框架，你会想要看看或者从这个社区已有并且提供了活跃支持的项目起步。表 5-2 列出了几种流行框架的起步项目。

表5-2：JavaScript MVC框架的启动项目

框架	起步项目
Backbone（http://backbonejs.org/）	Backbone Boilerplate（https://github.com/backbone-boilerplate/backbone-boilerplate）
Angular（http://angularjs.org/）	Angular Seed（https://github.com/angular/angular-seed）
Ember（http://emberjs.com/）	Ember Starter Kit（https://github.com/emberjs/starter-kit）

这些框架内容十分宽泛，不可能在这里做出全面的介绍。要深入了解请参见 O'Reilly 的 Angular（http://oreil.ly/angularJS）和 Backbone（http://oreil.ly/dev_backbone_js_apps）。

这些框架也并非唯一选择。JavaScript 框架有自己的依赖，也影响了一些库去扩展它们的核心功能。比如 jQuery 就是许多项目的先决条件，而 underscore.js 则是 Backbone 的一个依赖。Backbone 开发者倾向于使用 require.js 来做脚本加载和代码组织；它还促发了一个 MV* 框架 Spine（http://spinejs.com/）。Angular-UI（https://angular-ui.github.io/）提供了用户界面组件。jQuery 促成了一整个相关库、插件和扩展的生态系统，其中像 jQuery UI（http://jqueryui.com/）这样的大型库提供了各种控件，还有像 TouchPunch（http://touchpunch.furf.com/）这样特定功能的小型库可用于触摸屏幕事件处理。

还有其他起始项目，它们将 JavaScript 库和其他以浏览器兼容性和响应式设计为目标的起始项目结合到了一起。有一个结合了 Angular UI 和 Bootstrap（https://angular-ui.github.io/bootstrap/）的项目，可以拿来与 jQuery UI 和 Bootstrap（https://github.com/jquery-ui-

bootstrap/jquery-ui-bootstrap）项目比较。

除了 MVC 框架，如果要写 jQuery 插件，你还可以使用 jQuery Boilerplate（http://jqueryboilerplate.com/）来配置合适的项目结构。如果你正在写样板代码并且确定他人也一定会遇到同样的问题，那就有必要上网找一找看是否已经有现成的起始项目。

5.4 获取起始项目

有几种不同的方法获取起始项目来帮助你开启开发之旅。

5.4.1 直接从仓库下载

大多数项目都维护在线上的公共源码库，通常是 GitHub（https://github.com/）。

GitHub 仓库“名人堂”

起始项目、推进响应式 Web 设计的资源以及 JavaScript 库都属于 Github 上最受欢迎的仓库（https://github.com/trending）。

- 响应式设计资源
  - Modernizr（https://github.com/Modernizr/Modernizr）
  - Normalize CSS（https://github.com/necolas/normalize.css）
- 起始项目
  - Bootstrap（https://github.com/twbs/bootstrap）
  - HTML5 Boilerplate（https://github.com/h5bp/html5-boilerplate）
- JavaScript 库
  - jQuery（https://github.com/jquery/jquery）
  - Backbone（https://github.com/documentcloud/backbone）
  - Foundation（https://github.com/zurb/foundation）
  - Angular（https://github.com/angular/angular.js）
  - Underscore（https://github.com/jashkenas/underscore）
  - Ember（https://github.com/emberjs/ember.js）
  - jQuery UI（https://github.com/jquery/jquery-ui）
  - Knockout（https://github.com/knockout/knockout）

5.4.2 从入门网站下载

类似 Initializr（http://www.initializr.com/）和 HTML5 Reset（http://html5reset.org/）的网站主要依赖源代码库，但是也包括了评论、对比、起始项目文档、常规开发和设计话题。

5.4.3　IDE生成的起始项目

类似 WebStrom（一个由 JetBrains 公司开发的商业项目）的集成开发环境，提供让用户从模板中创建新项目的选项，如图 5-2 所示。WebStrom 包括本章讨论的一些项目，以及 Node.js（Node.js boilerplate，https://github.com/robrighter/node-boilerplate）和 Node.js express（http://expressjs.com/）和 Dart（https://www.dartlang.org/）起始项目。



图 5-2：客户端起始项目

5.5　前端工程师的崛起

显然，现在的 Web 开发已经比早期复杂得多了。不具备也不关注关键开发技能的设计师不可能跟上最新的发展。同样的，服务端开发者也大多不具有成熟的设计能力，对客户端的最新发展也不了解。这就促使了一个新职业的崛起：*前端工程师*。如果讨论的这些话题还不足以使你相信新信息的大爆炸，那么就请考虑下与客户端开发和流程相关的一些细微差别。

5.5.1　客户端模板

前面介绍的一些 JavaScript 框架内置了 JavaScript 模板方案。还有其他各种各样的独立模

板，很多都可以和框架的内置模板任意替换。LinkedIn（http://linkd.in/1dnFOzZ）的工程师在决定使用 dust.js（http://akdubya.github.io/dustjs/）之前调研了多达 26 种客户端模板技术。这个领域还有一些吸引人的发展，例如能够在需要时将失效备援的任务交给服务端的客户端 JavaScript 模板。

5.5.2 资源管道

老式的 Web 开发只是简单地编辑并引用自己 Web 服务器上的相关资源。而现在资源可以由外部的*内容分发网络*（CDN）提供。另外，资源不再是简单地编辑和引用，而是通常在做了各种预处理之后才放到 Web 服务器上提供服务。*资源管道*可用于预编译、合并，以及压缩可用的 Web 资源，并实现这些资源的缓存管理。

在过去的几年，Ruby 社区出现了资源编译器。早期的例子是 Jammit（http://documentcloud.github.io/jammit/）和 Sprockets（https://github.com/sstephenson/sprockets）。后来，管道资源被纳入到 Rails 并且被其他语言的 Web 框架采用，例如 Java's Play2（https://www.playframework.com/documentation/2.0/Assets）。

资源管道可用于多项任务。有的可以用在提供较大文件时减少网络延时。JavaScript 和 CSS 文件可以被精简（去除空格和多余字符）、合并（减少网络总请求次数）和压缩（使用 gzip 或其他压缩算法）。还有些资源管道会用于与缓存相关的任务。例如，`Play` 框架使用 ETag HTTP Headers（http://en.wikipedia.org/wiki/HTTP_ETag）追加一个由资源名称和文档最后改变日期生成的值，并提供了设置 Cache-Control 头信息的选项。

另外，预处理步骤可以把其他语言编译成 JavaScript（例如 CoffeeScript 和 Dart，分别参见 http://coffeescript.org/ 和 https://www.dartlang.org/）。这也给了那些不太喜欢 JavaScript 这门语言本身的人一些选择。

对 CSS 也可以进行预处理，这样做可以减少重复代码。重复的代码越少应用越容易维护，当然这需要增加额外的编译步骤。预编译器需要在被 HTML 页面引用之前处理原始文件，并解析代码中的引用以生成标准的样式表。以下是 CSS 编译器的几种预处理类型：

- 在整个样式表中定义可以替换的变量值（例如，在许多 CSS 类中定义的颜色值）；
- 创建接受不同参数的方法为类赋值；
- 实现 CSS 类的继承。

CSS 处理器起源于 Ruby 开发者，但是已经在 Java 社区获得认可。LESS CSS Maven 插件（http://mojo.codehaus.org/lesscss-maven-plugin/）就是一个例子。

在资源管道中实施预编译步骤多少有些争议。对那些从来不是只处理硬编码 CSS 的设计师来说，预处理器可能带来的编程工作会令他们感到不安。尽管开发者可能更接受预处理器

的概念（基于它们在其他编程环境中的使用），但是改变任何人的工作流程都是令人不安的。这就需要前端开发工程师这样一个新角色来填补开发者和设计师之间的空白，他们需要独特的工作流以及合适的工具集来展开他们的职责工作。

5.5.3 开发流程

Yeoman（http://yeoman.io/）是一个基于 node 的包，它利用三个工具提供了一套开发流程。这三个工具是：yo（https://github.com/yeoman/yo，用于脚手架）、grunt（http://gruntjs.com/，用于编译）和 bower（http://bower.io/，用于包管理）。其他基于 node 的包例如 karma（用于测试）和 Docco（用于生成文档）也可用于开发和构建流程。

5.6 项目

有兴趣了解最简单框架示例的价值，以及框架与功能完备的起始项目之间的关系，可以考虑下面使用 Angular 的示例：

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
  </head>
  <body>
        Angular Expression 1 + 2 evaluates to:  {{ 1 + 2 }}
  </body>
</html>
```

Angular 框架的作用是显而易见的，因为在文档中有很多 XML 属性（称为指令），它们并不是文档中标准 HTML 的内容。在这个例子中，ng-app（https://docs.angularjs.org/api/ng/directive/ngApp，在 HTML 标签中）用于自动引导应用。这类属性在一个 HTML 页面中只能出现一次。它指定了应用的根，并可以选择性地指定一个模块名（尽管在这个例子中它是空的）。script 标签说明使用了 Angular；还可以包括提供了其他辅助功能的 Angular 脚本。最终，Angular 中可见的功能会通过表达式（http://docs-angularjs-org-dev.appspot.com/guide/expression）展现。Angular 表达式是类似于 JavaScript 的代码片段，它被放置于两对花括号之间解析成输出。

这是一个很简单的只对表达式求值的示例。它甚至都没有演示出 Angular 的 MV* 框架特性，因为例子中没有包括控制器和数据绑定。它仅仅说明了创建一个 Angular 应用所必备的最小特征。下面这个例子包括了模型和控制器：

```
<!doctype html>
<html ng-app>
  <head>
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.6/angular.min.js">
```

```
</script>
    <script>
        function HelloCntl($scope) {
                        $scope.name = 'World';
                }
    </script>
  </head>
  <body>
    <div ng-控制器="HelloCntl">
      Your name: <input type="text" ng-模型="name" />
      <hr/>
      Hello {{name}}!
    </div>
  </body>
</html>
```

在 Angular 中 一 个 控 制 器（https://docs.angularjs.org/guide/dev_guide.mvc.understanding_controller）其实就是一个函数，用于在指定的作用域内实现行为。在这个例子中，控制器用于把模型（https://docs.angularjs.org/guide/dev_guide.mvc.understanding_model，`name` 变量）绑定到输入框。输入框中值的变化会立即反映到表达式中。作为一个框架，提供远程 Ajax 服务调用是非常重要的。最终示例如下：

```
<!doctype html>
<html ng-app="GoogleFinance">
  <head>
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.6/angular.min.js">
</script>
    <script src="http://code.angularjs.org/1.0.6/angular-resource.js"></script>
    <script>
      angular.module('GoogleFinance', ['ngResource']);

      function AppCtrl($scope, $resource) {
        $scope.googleFinance = $resource('https://finance.google.com/finance/info',
                  {client:'ig',q: 'AAPL',callback:'JSON_CALLBACK'},
                  {get: {method:'JSONP',isArray: true}});

          $scope.indexResult = $scope.googleFinance.get();

          $scope.doSearch = function () {
            $scope.indexResult = $scope.googleFinance.get({q: $scope.searchTerm});
          };
      }
    </script>
  </head>
<body>
  <div ng-控制器="AppCtrl">
    <form class="form-horizontal">
      <input type="text" ng-模型="searchTerm">
      <button class="btn" ng-click="doSearch()">
        Search
      </button>
          </form>
```

```
        Current Price: {{indexResult[0].l_cur}}<br/>
        Change:        {{indexResult[0].c}}<br/>
    </div>
  </body>
  </html>
```

Angular resource（https://docs.angularjs.org/api/ngResource/service/$resource）用于进行后端的调用。需要引入其定义脚本，而且需要在控制器中作为参数传递。因为这是一个跨域调用，所以用到了 JSONP。`searchTerm` 模型通过谷歌财经 API 来搜索股票代码，返回结果是一个 JSON 数组，并且当前价格和变更的字段会显示在检索结果中。

这三个例子有助于教学和示范。它们做了简化以强调哪些功能是必需的，以及它们之间的关系。

添加 Bootstrap 的样式表和一些基本风格样式也并不费事。本章的代码与之前的例子相比增加了一些额外的变化，这是通过引入 `bootstrap.css`、创建容器类和一些视图行并增加搜索图标和调整设计来实现的。这些东西全都在同一个文件里面，一个标准的起始项目会将这些资源打散并用标准的目录结构组织它们。

通过将上述代码的相关部分与索引页（或相关视图）相关联，可以将这些例子添加进类似 AngularJS 或 Bootstrap 的起始项目中。理想情况下，应该将这些例子中的内置 JavaScript 代码提取出来写入独立的文件（将模块写入 `app.js`，将控制器写入 `controllers.js`）。

5.7　小结

在任一成功的开发团队中，每个成员都利用其独特的优势共同为项目的效益做贡献。一个成员对自己领域的专注使其他成员能够密切关注其他领域，这就有效地将问题控制在他们的关注范围内。起始项目和 JavaScript 框架使得开发者不用关心 BOM 和 DOM 的实现细节、浏览器兼容性、初始代码组织等问题和其他挑战。

硬件的提升和 JavaScript 性能的优化使我们能够创建大型 JavaScript 应用。当使用标准的设计，并通过减少 Web 开发项目中的重复任务来专注于项目重点时，大型应用的管理就会变得更加轻松。本章所介绍的这些框架可以帮助你快速地创建并启动你的项目，并使你的项目成员可以立即专注于功能的实现。

第6章

Java Web API服务器

“一个只关心自己的人不足以成大器。”

——本杰明 • 富兰克林

将可重用的组件打包这一点让扩展编程语言的能力成为可能。对于一门特定的语言，打包会影响到部署方式。打包需要考虑标准命名规范、元数据文件、数字签名、代码混淆、代码管理、包含相关的文件 / 资源，以及压缩机制。

打包会影响部署。事实上，一门语言提供的部署选项能左右一个项目的结构和开发流程，在 Java 中尤其如此。Java 源文件的名字反映了其包含的公有类的名字；Java 的包结构遵循文件系统中的目录结构。Java 使用了多个不同类型的包格式。一般的代码被打包到 *Java 归档文件*（JAR），Web 应用被打包到 *Web 应用归档文件*（WAR），一组相关的 Web 应用被打包到*企业级应用归档文件*（EAR）。WAR 可以被部署到 servlet 容器中，而 EAR 需要完整企业级 Java 的支持，必须要有类似 JBoss 这样的应用服务器。

Java 的打包模型有很多优点，但却限制了部署的可能性。从 Java 发展的初期，标准 Web 应用的开发实践就要求独立安装和配置应用服务器。然而实际上这些并不是部署 Java Web 应用所必需的。即使使用了一个 Web 容器，也有可能在开始开发前避免安装和配置它所带来的开销。

6.1 更简单的服务器端解决方案

客户端技术的爆发势不可挡，这源于将浏览器作为平台所带来的挑战、JavaScript 语言和为了管理复杂性而开发出来的 JavaScript 类库。另一复杂性来自和设备相关的开发，它可

以是移动设备上的 Web 应用，也可以是通过 HTTP 调用 Web 服务的原生应用。然而，可喜的是在这种新的客户端 – 服务器端的 Web 应用模型里，服务器端的代码被简化了。

传统上，Web 应用是一个相当大的结构，它可以跑在应用服务器上、Web 容器里或者 Web 服务器上。写作此书时，值得尊敬的 Apache Web 服务器（https://www.openhub.net/p/apache）拥有超过 200 万行代码，需要成百上千的人年[1]才能完成。J2EE 开发的复杂性声名狼藉（以致在最近版本的 JEE 中对其进行了大幅地简化）。服务器端的模型 – 视图 – 控制项目（Struts）和依赖注入框架（Spring）提供了组织功能，减轻了配置的负担，同时突出了开发 Java Web 应用时对服务器端所要求的规模和可伸缩性。

客户端 – 服务器端架构的 Web 应用使用的服务器不维持会话状态，大多由简单的 API 组成。尽管它们可以利用中间件支持 Web 服务，但它们常常不这样做。直接或间接受到来自脚本语言世界的影响，现在建立一个最小化的服务器非常简单，它可以为前端开发者提供必需的功能，让他们在没有一个完整的服务器端实现的条件下也能工作。Java 和其他基于 JVM 的语言为开发者提供了很多可能性，开发者可以在一个项目里包括所有服务器端的功能，而不需要安装、配置和维护一个完整的服务器实现。和以前相比，这为开发流程提供了截然不同的可能性。除了开发时会受益，这种轻量级的服务器解决方案也让将应用部署到云环境下时的水平扩展变得简单了。

使用 Java 或利用一些类库就可以创建“无容器”的 Web 应用服务器。图 6-1 展示了本章用到的服务器和它们之间的关系。图上方的类库基础设施更稳固，理论上对适合某领域的框架来说，它们是个更高效的起点。这里并没有列出所有的服务器，但可以从多维度对其扩展。比如各种基于 JVM 的语言都有对应的 Web 框架（JRuby 有 Sinatra、Rails，Jython 有 Django）。Typeset（http://typesafe.com/）是一个基于 Play 的 Scala 框架。vert.x（http://vertx.io/）尤为特别，它支持多种 JVM 语言。

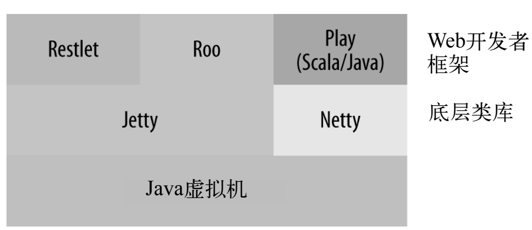


图 6-1：服务器类库

本章的例子限于以 Java 为中心的工作流和构建流程。基于 JVM 的语言根据其起源不同，有不同的工作流实践。Groovy 和 Scala 的初始目标平台就是 JVM，而 Ruby 和 Python 是后

注 1：表示人口生存时间长度的复合单位，是人数同生存年数乘积的总和。例如一个人生存一年是 1 人年，两个人各生存半年也等于 1 人年，这三个人共生存 2 人年。

来移植到 JVM 的语言。Groovy（和 Scala 中的少数）使用的技术和实践有迹可寻，Java 开发者很熟悉其中的模式。JRuby 和 Jython 起源不同，它们的方式对 Java 开发者来说就不那么熟悉了（但是那些从其他实现迁移到 JVM 的开发者却认可这种方式）。这就要求读者使用各自社区特有的开发安装流程、工具链和命令。

6.2 基于Java的服务器

可以单独使用 Java，或者使用一些高级的类库编写服务器端。很多时候使用的是一种基本的模式：服务器类将请求委托给一个处理器，处理器继承了一个抽象类或实现了一个接口，该类或接口定义了一个 `handle` 方法。图 6-2 展示了服务器类和处理器类之间的基本关系。具体的处理器实现了 `handle` 方法，在下面的例子中，处理器返回一个 JSON 对象。

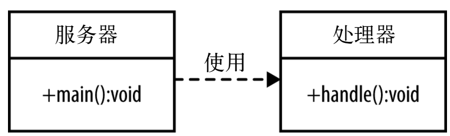


图 6-2：服务器 – 处理器类图

关于例子

和本书其他一些例子类似，我们对本章的例子有意做了简化，让其和当下讨论的主题相关。比如，很多 JSON 响应都是简单的字符串。在现实中，更常见的情况是使用 Jackson 或其他类库来构建响应。大多数开发者都会工作在高级的、功能完整的框架之上，但使用字符串能将书中出现代码的含义表达得更清楚。在实际项目中使用类库和注释看起来整齐、高效、意义明确，但在示例代码中会隐藏过多的功能，代码即使在最好的情况下看起来也不会很清楚。

这些例子的意义在于展示不总是*需要*应用服务器，编写可以工作的、有特殊用途的服务器是一件非常简单的任务。很多开发选择使用高级的框架，但是知道底层的类库是怎样工作的以后，能帮助调试和分析堆栈错误跟踪信息和日志。

6.2.1 Java HTTP服务器

Java 标准版附带了一个 HTTP 服务器 API，可以用它来创建一个简单、能工作的嵌入式 HTTP 服务器。下面的程序不需要任何外部依赖。它监听 8000 端口，对进来的请求返回一个成功的 HTTP 响应（200），响应内容为如下的 JSON 对象：

```
{"testResponse":"Hello World"}
```

这里需要一个 Java 源文件，包含一个定义了 `main` 方法的类。在 `main` 方法中，创建了一个服务器实例，该服务器实例根据 URL 将请求委托给一个静态处理器（这里被定义为一个静态内部类）：

```
import java.io.*;
import com.sun.net.httpserver.*;
import java.net.InetSocketAddress;

public class HttpJsonServer {

    public static void main(String[] args) throws Exception {

        HttpServer server = HttpServer.create(
           new InetSocketAddress(8000), 0
        );
        server.createContext("/", new MyHandler());
        server.setExecutor(null);
                System.out.println("Starting server on port: 8000");
        server.start();
    }

    static class MyHandler implements HttpHandler {

        public void handle(HttpExchange t) throws IOException {
            String response = "{\"testResponse\":\"Hello World\"}";
                        t.getResponseHeaders().set(
                "Content-Type",
                "application/json"
            );
            t.sendResponseHeaders(200, response.length());
            OutputStream os = t.getResponseBody();
            os.write(response.getBytes());
            os.close();
        }
    }
}
```

这个例子非常简单，没有外部依赖，也不需要构建脚本，只需要编译运行即可：

```
$ javac HttpJsonServer.java
$ java HttpJsonServer
Starting server on port: 8000
```

可以在另外一个命令行会话里使用 Curl 访问服务器：

```
$ curl -i http://localhost:8000
HTTP/1.1 200 OK
Content-type: application/json
Content-length: 30
Date: Sun, 09 Jun 2013 01:15:15 GMT

{"testResponse":"Hello World"}
```

显然这不是一个功能完备的例子，但是它表明使用 Java 开发一个能工作的 HTTP 服务器是一个相对简单可实现的目标。

6.2.2 Jetty嵌入式服务器

Jetty（http://www.eclipse.org/jetty/）是一个基于 Java 的 HTTP 服务器和 Java Servlet 容器，它由 Eclipse 基金会维护。除了自身是 Eclipse IDE 的一部分，它还被用在很多产品中，比如 ActiveMQ、Maven、Google App Engine 和 Hadoop。它不仅仅是一个 Web 服务器，它还支持 Java Servlet API、SPDY 和 WebSocket 协议。因为这么多项目中都用到了它，很多开发者至少知道它的存在和功能。但是，很少人知道它可以容易地作为一个组件嵌入一个定制的 Java 服务器。

基于 Jetty 的服务器需要包含外部模块，因此需要一个构建脚本。在一本书中描述 Maven 配置的挑战之一是 pom.xml 太过庞大冗长。Gradle 配置相对更简短扼要。

下面的 Gradle 配置包括了 Jetty，而且增加了一个启动服务器的任务：

```
apply plugin:'java'

repositories{mavenCentral()}
dependencies{compile 'org.eclipse.jetty:jetty-server:8.1.0.RC5'}

task(startServer, dependsOn: 'classes', type: JavaExec) {
    main = 'com.saternos.embedded.TestJettyHttpServer'
    classpath = sourceSets.main.runtimeClasspath
    args 8000
}
```

其遵循的模式和单纯基于 Java 的例子类似，定义了一个处理器类返回一个 JSON 响应：

```
package com.saternos.embedded;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.eclipse.jetty.server.Request;
import org.eclipse.jetty.server.handler.AbstractHandler;

public class JsonHandler extends AbstractHandler
{
    public void handle(
        String target,
        Request baseRequest,
        HttpServletRequest request,
        HttpServletResponse response
    )
        throws IOException, ServletException
    {
```

```
            response.setContentType("application/json;charset=utf-8");
            response.setStatus(HttpServletResponse.SC_OK);
            baseRequest.setHandled(true);
            response.getWriter().println(
                "{\"testResponse\":\"Hello World\"}"
            );
        }
    }
```

main 方法创建了一个服务器实例，服务器对象将接收到的请求委托给处理器：

```
package com.saternos.embedded;

import org.eclipse.jetty.server.Server;

public class TestJettyHttpServer
{
    public static void main(String[] args) throws Exception
    {
        Server server = new Server(Integer.parseInt(args[0]));
        server.setHandler(new JsonHandler());
        System.out.println("Starting server on port: " + args[0]);
        server.start();
        server.join();
    }
}
```

使用 Gradle 构建和运行服务器：

```
$ gradle build startServer
```

通过 Curl 得到的响应正如你所期待的那样，同时返回一个引用，指明服务器是 Jetty：

```
$ curl -i http://localhost:8000
HTTP/1.1 200 OK
Content-Type: application/json;charset=utf-8
Content-Length: 31
Server: Jetty(8.1.0.RC5)

{"testResponse":"Hello World"}
```

如果你倾向于从头开始编写一个定制 API 的服务器，Java 自带的基本模块或者像 Jetty 这样的扩展类库就足够了。很多和编写 Web API 相关的项目都在内部使用了 Jetty，比如 Restlet。

6.2.3 Restlet

Restlet API（http://restlet.com/）是直接按照 REST 的概念设计的，资源、表示、连接器、组件和媒体类型。使用该框架可自然地实现 REST API。尽管下面的例子不是纯 RESTful 的，但依然展示了只需要很少的一点代码就能创建一个返回 JSON 的服务器：

```
package com.saternos.embedded;

import org.restlet.*;
import org.restlet.data.*;
import org.restlet.resource.*;
import org.restlet.ext.json.JsonConverter;

public class TestRestletHttpServer extends ServerResource {

    public static void main(String[] args) throws Exception {
        Component component = new Component();
            component.getServers().add(Protocol.HTTP, Integer.parseInt(args[0]));
            component.getDefaultHost().attach("/", TestRestletHttpServer.class);
            component.start();
        }

        @Get ("json")
        public String getGreeting() {
            return "{\"testResponse\":\"Hello World\"}";
        }
}
```

6.2.4 Roo

Roo（http://www.bit.ly/MfMxBL）让你几分钟之内就能创建一个项目。新建的项目运行在一个嵌入的 Jetty（或 Tomcat）服务器上，同时还有一个内存数据库（你可以设计自己的数据模型，他们能自动传递给选择的数据库）。因此不需要花费一点时间配置服务器，只要在 REPL 上键入命令，你就能创建一个包含 Web 服务器和数据库的传统 Web 应用。

请下载最新版本的 Roo（http://www.bit.ly/1dKJ9EN），解压并将 `roo.sh`（或 `roo.bat`）加入系统路径。执行 `roo.sh`，系统会提示你完成初始化安装（接受许可协议）。随时可输入 `help` 或者 `hint` 给出可用选项的提示：

```
$ roo.sh
    ____  ____  ____
   / __ \/ __ \/ __ \
  / /_/ / / / / / / /
 / _, _/ /_/ / /_/ /
/_/ |_|\____/\____/    1.2.4.RELEASE [rev 75337cf]

Welcome to Spring Roo. For assistance press TAB or type "hint" then hit ENTER.
roo>
```

Roo 的命令行像一个创建基于 Spring 的 Web 应用的专家系统。在开发时，可随时根据需要键入 `hint` 命令获取可用选项。系统会带领你建立数据存储、定义实体和成员，最终创建一个 Web 应用。

需要注意的是，有时需要跳出命令行工作。比如，如果你键入错误的实体或成员，需要将它们从生成的代码中删除时。

在 Roo 中执行下面一组命令，会创建一个标准的包含 JSON 服务的 JEE Web 应用（这里省略掉了每条命令的输出）：

```
project com.saternos.bookshop

jpa setup --provider HIBERNATE --database HYPERSONIC_IN_MEMORY

entity jpa --class ~.domain.Author --testAutomatically
field string --fieldName name --sizeMin 2 --notNull

entity jpa --class ~.domain.Book --testAutomatically
field string --fieldName name --notNull --sizeMin 2
field number --fieldName price --type java.math.BigDecimal
field set --fieldName authors --type ~.domain.Author

json all --deepSerialize
web mvc json setup
web mvc json all

web mvc setup
web mvc all --package ~.web
```

该应用对一列图书提供了 CRUD 操作。在项目初始化后，配置了一个内存数据库，然后创建了两个实体（和对应的成员）。接下来的三条命令通过 JSON Web API 向外暴露出实体，最后的两条命令生成了一个基于实体的 JEE 管理应用。

该应用构建后可以运行在 Jetty 上：

```
mvn clean install jetty:run
```

可以通过 Curl 调用 JSON API 创建作者对象，第一个例子展示了如何创建一位作者，第二个展示了如何创建一组作者。

```
curl -i -X POST -H "Content-Type: application/json" \
-H "Accept: application/json" \
-d '{name: "Simon St. Laurent"}' \
http://localhost:8080/bookshop/authors

curl -i -X POST -H "Content-Type: application/json" \
-H "Accept: application/json" \
-d '[ {name: "Douglas Crockford"}, {name: "Michael Fogus"} ]' \
http://localhost:8080/bookshop/authors/jsonArray
```

你可以通过 http://localhost:8080/bookshop/ 访问该应用的主页，该页面拥有系统默认的专业设计风格，如图 6-3 所示。

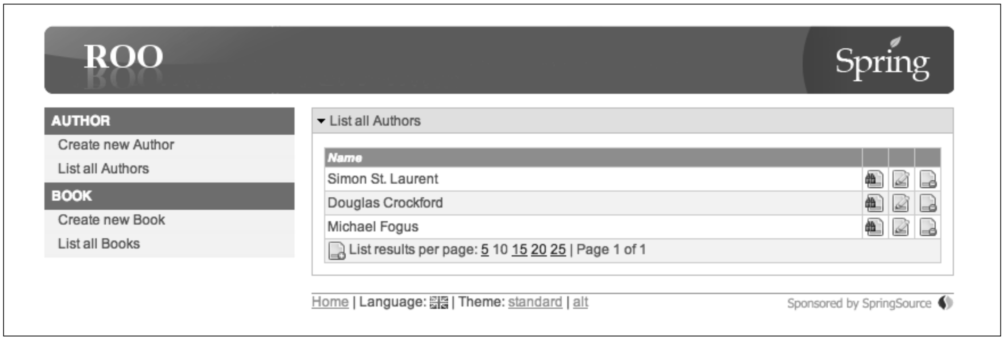


图 6-3：使用 Roo 创建的 Web 应用

同样的 URL，使用 HTTP `delete` 操作可以删除一条记录：

```
curl -i -X DELETE -H "Accept: application/json" \
http://localhost:8080/bookshop/authors/1

curl -i -X DELETE -H "Accept: application/json" \
http://localhost:8080/bookshop/authors/2

curl -i -X DELETE -H "Accept: application/json" \
http://localhost:8080/bookshop/authors/3
```

可以通过嵌入作者的引用来一起添加图书连同其作者。图 6-4 展示了执行下述 Curl 命令[2]后“List all Books”页面的变化：

```
curl -i -X POST -H "Content-Type: application/json" \
-H "Accept: application/json" \
-d '{name:"JavaScript: The Good Parts",'`
    `'price:29.99,'`
    `'authors:[{name: "Douglas Crockford"}]}' \
http://localhost:8080/bookshop/books

curl -i -X POST -H "Content-Type: application/json" \
-H "Accept: application/json" \
-d '{name:"Functional JavaScript",'`
  `'price:29.99,'`
  `'authors:[{name: "Michael Fogus"}]}' \
http://localhost:8080/bookshop/books

curl -i -X POST -H "Content-Type: application/json" \
```

注 2：将代码格式化为人们易读的形式很有挑战性。这是因为计算机和人类解释空格和换行的方式不同——还有本书同时以纸质和电子版出版的事实。在实际中，上述 `curl` 命令最好没有换行，而且空格的数量也越少越好。遗憾的是，这样会让页面上的文字自动换行显示错误或者文字超出页面的边框。在 bash 会话中，长命令可通过在每行末尾输入反斜杠（\）分成多行。我们将在本书中沿用这种风格，尽管它不是在任何情况下都管用。上述例子中将 JSON 字符串划分成多行，使用反引号（`）忽略了插入的空格和换行符。我们利用 bash 这个小技巧是为了方便那些希望在本书电子版上复制命令的人。

```
-H "Accept: application/json" \
-d '{name:"Introducing Elixir",'`
  `'price:19.99,'`
  `'authors:[{name: "Simon St. Laurent"}]}' \
http://localhost:8080/bookshop/books
```

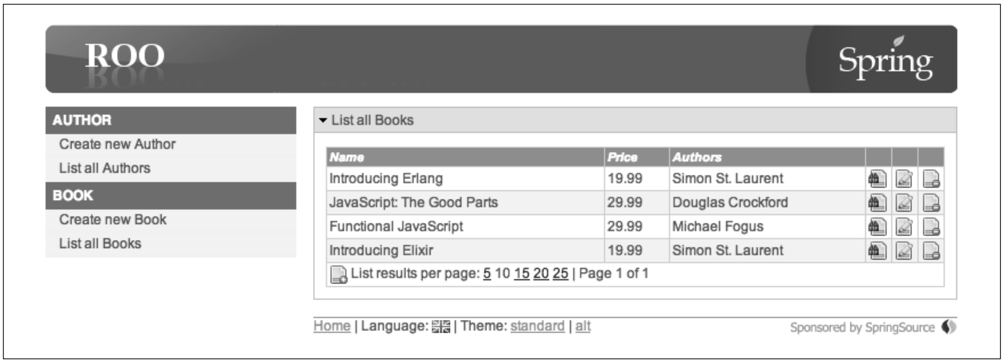


图 6-4：使用 Roo 创建的 Web 应用

获取插入的作者和图书列表：

```
curl http://localhost:8080/bookshop/authors
curl http://localhost:8080/bookshop/books
```

可以使用 ID 访问数组中的一个对象：

```
curl http://localhost:8080/bookshop/books/1
```

在命令行中格式化 JSON

有多种对从命令行返回的 JSON 进行格式化的方法。可以将输出重定向到一个文件中，然后使用支持 JSON 格式化的编辑器打开。还有特殊用途的工具，比如 jq（http://stedolan.github.io/jq/），可以格式化通过管道传入的 JSON。如果你安装了 Python，格式化就非常简单：

```
curl http://localhost:8080/bookshop/books | python -mjson.tool
```

有了 Web API，就可以在使用 Roo `scaffolding` 命令创建 jspx 页面的地方创建（或增加）标准的 HTML 页面。创建一个名为 /src/main/webapp/test.html 的文件，并可以通过 http://localhost:8080/bookshop/test.html 访问它。获取图书列表的 `GET` 操作可使用 jQuery 的 `getJSON` 方法调用：

```
<html>
<head>
  <title>Book List</title>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.10.1/jquery.min.js">
```

```
</script>
</head>
<body>

  <h3>Books</h3>
  <ul class='booklist'></ul>
<script>
  (function() {
    $.getJSON('/bookshop/books', function(books) {
      $.each(books, function(nil,book) {
        $('.booklist').append('<li>' + book['name'] + '</li>');
      });
    });
  })();
</script>
</body>
</html>
```

因其可交互的特性，Roo 是市面上最易上手的工具之一。如果你想获取更多关于该框架的介绍，可免费下载（http://spring-roo-repository.springsource.org/Getting_Started_with_Roo.pdf）*Getting Started with Roo*（http://shop.oreilly.com/product/0636920020981.do，O'Reilly 出版）一书，而 *Spring in Action*（http://www.manning.com/rimple/，Manning 出版）一书也提供了很多关于 Roo 的信息。

6.2.5 Netty嵌入式服务器

Jetty 是和 Eclipse 基金会相关的，但 JBoss 社区也出现了一个某些功能与其重合的项目。Netty（http://netty.io/）被用来创建优化网络应用（服务器和客户端协议）。优化网络应用在扩展性上比一般应用要好，Twitter（https://blog.twitter.com/2011/twitter-search-now-3x-faster）采用 Netty 去解决它们的性能问题就表明了 Netty 的成熟，和其在扩展性上的能力。

使用 Netty 的例子和前面展示的使用 Jetty 的例子类似，它们都遵循类似的模式：一个包含 `main` 方法的类初始化并启动服务器，一个处理器类负责处理进来的请求：

```
package com.saternos.embedded;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.socket.nio.*;

public class TestNettyHttpServer {

    public static void main(String[] args) throws Exception {

                ServerBootstrap bootstrap = new ServerBootstrap();

                bootstrap.group(new NioEventLoopGroup(), new NioEventLoopGroup())
                                .channel(NioServerSocketChannel.class)
                                .localAddress(Integer.parseInt(args[0]))
                                .childHandler(new JsonServerInitializer());
```

```
            System.out.println("Starting server on port: " + args[0]);
        bootstrap.bind().sync().channel().closeFuture().sync();
    }
}
```

处理器类继承自 ChannelInboundMessageHandlerAdapter，该类被包含在服务器的管道里。messageReceived 方法拿到请求并创建响应：

```
package com.saternos.embedded;

import io.netty.buffer.Unpooled;
import io.netty.channel.*;
import io.netty.handler.codec.http.*;
import io.netty.util.CharsetUtil;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Locale;
import java.util.TimeZone;

public class JsonHandler extends ChannelInboundMessageHandlerAdapter<HttpRequest> {

    public void messageReceived(ChannelHandlerContext channelHandlerContext,
                                HttpRequest httpRequest) throws Exception {
        StringBuffer buf = new StringBuffer();
        buf.append("{\"testResponse\":\"Hello World\"}");

        SimpleDateFormat dateFormatter = new SimpleDateFormat(
                                          "EEE, dd MMM yyyy HH:mm:ss zzz",
                                          Locale.US
                                         );
                dateFormatter.setTimeZone(TimeZone.getTimeZone("GMT"));
                Calendar time = new GregorianCalendar();

        HttpResponse response = new DefaultHttpResponse(HttpVersion.HTTP_1_1,
                                                  HttpResponseStatus.OK);

response.setHeader(HttpHeaders.Names.CONTENT_TYPE,
                   "application/json;charset=utf-8");
response.setHeader(HttpHeaders.Names.CONTENT_LENGTH, buf.length());
response.setHeader(HttpHeaders.Names.DATE,
                   dateFormatter.format(time.getTime()));
response.setHeader(HttpHeaders.Names.SERVER,
                   TestNettyHttpServer.class.getName() +
                   ":io.netty:netty:4.0.0.Alpha8");
response.setContent(Unpooled.copiedBuffer(buf, CharsetUtil.UTF_8));
channelHandlerContext.write(response).addListener(
                    ChannelFutureListener.CLOSE);
    }
}
```

这里还需要一个实现 ChannelInitializer 接口的类来建立管道。

`initChannel` 方法引用了 Channel 接口（http://netty.io/4.0/api/），它被用来创建和访问 / SocketChannel（(http://netty.io/4.0/api/)）管道。管道转换或更改传进来的值。`HttpRequestDecoder`（http://netty.io/4.0/api/io/netty/handler/codec/http/HttpRequestDecoder.html）接收原始字节，并用它们构建 HTTP 相关的对象，`HttpChunkAggregator` 使用分块传输编码方式归一化请求。随后的两个管道类对响应进行编码和分块，然后传递给实例应用具体的处理器类（这里生成头信息和实际的响应）：

```
package com.saternos.embedded;

import io.netty.channel.*;
import io.netty.channel.socket.SocketChannel;
import io.netty.handler.codec.http.*;
import io.netty.handler.stream.ChunkedWriteHandler;

public class JsonServerInitializer extends ChannelInitializer<SocketChannel> {

    public void initChannel(SocketChannel socketChannel) throws Exception {
        ChannelPipeline pipeline = socketChannel.pipeline();
        pipeline.addLast("decoder", new HttpRequestDecoder());
        pipeline.addLast("aggregator", new HttpChunkAggregator(65536));
        pipeline.addLast("encoder", new HttpResponseEncoder());
        pipeline.addLast("chunkedWriter", new ChunkedWriteHandler());
        pipeline.addLast("handler", new JsonHandler());
    }
}
```

构建和执行上述应用会得到和前一个例子类似的响应：

```
$ gradle build startServer
$ curl -i http://localhost:8000

HTTP/1.1 200 OK
Content-Type: application/json;charset=utf-8
Content-Length: 30
Date: Fri, 14 Jun 2013 14:02:57 GMT
Server: com.saternos.embedded.TestNettyHttpServer:io.netty:netty:4.0.0.Alpha8

{"testResponse":"Hello World"}
```

Netty 是一个相对底层、通用的网络类库。这为创建优化应用提供了很多灵活性，其代价是要建立额外的管道和处理器。上述例子有点长，而且也更复杂，但是理解它是值得的，因为 Netty 是其他几个高级框架，比如 Play 和 Vert.x 的基础。Norman Maurer 是 Netty 的核心开发者之一，他编写了 *Netty in Action*（http://www.manning.com/maurer/，Manning 出版）一书，该书深入阐述了这个复杂框架的功能细节。

6.2.6 Play服务器

Play 框架（http://www.playframework.org/）是一个轻量级、无状态的框架，非常适合创建

服务器 API。使用一些简单的命令就能生成一个 Play 应用，路由文件指明了 HTTP 操作和引用控制器方法的路径。代码可以使用 Java 或 Scala 编写，其中来自 Scala 社区的印记随处可见，比如使用 SBT 和基于 Scala 的模板。对 Play 项目的更改会立即反映在浏览器中，不需要再单独构建。

安装好 Play（https://www.playframework.com/documentation/2.1.1/Installing）后，使用 `play new` 创建一个新的 Play 应用，给应用起个名字，选择一个 Java 模板：

```
$ play new play-server
       _            _
 _ __ | | __ _ _  _| |
| '_ \| |/ _' | || |_|
|  __/|_|\____|\__ (_)
|_|            |__/

play! 2.1.1 (using Java 1.7.0_21 and Scala 2.10.0), http://www.playframework.org

The new application will be created in /Users/cs/Desktop/tmp/play-server

What is the application name? [play-server]
>

Which template do you want to use for this new application?

  1             - Create a simple Scala application
  2             - Create a simple Java application

> 2
OK, application play-server is created.

Have fun!
```

该命令生成的应用结构和 Ruby on Rails 应用相仿。后续的命令和修改文件操作都在新建目录下：

```
cd play-server
```

在这个例子中，给定一个 URL（`/json`），该应用返回 JSON。修改 `conf/routes`，为其加入下面一行来定义该 API：

```
GET     /json                       controllers.Application.json()
```

最后，修改路由文件中引用的 Java 控制器代码。

需要注意的是，区别于之前通过 JSON 字符串返回一个“Hello World”的例子，该例子中它创建了一个对象，在响应中将其序列化为 JSON（底层使用了 Jackson）。

在 JAX-RS（JEE Java API for RESTful）Web 服务中也用到了这种创建对象，然后在响应中将其序列化为目标格式的流程：

```
package controllers;

import play.*;
import play.mvc.*;

import views.html.*;

import org.codehaus.jackson.JsonNode;

import play.libs.Json;
import org.codehaus.jackson.node.ObjectNode;
import org.codehaus.jackson.node.ArrayNode;

// 优化打印
import org.codehaus.jackson.map.ObjectMapper;

public class Application extends Controller {

    public static Result index() {
        return ok(index.render("Your new application is ready."));
    }

    public static Result json() {

            ObjectNode result = Json.newObject();

                // 如何嵌套JSON对象
                ObjectNode child = Json.newObject();
                child.put("color", "blue");
                result.put("NestedObj",child);

                // 添加字符串
                result.put("status", "OK");
                result.put("something", "else");

                // 添加整数
                result.put("int", 1);

               // 添加JSON数组
           ArrayNode collection = result.putArray("coll");
               collection.add(1);
               collection.add(2);
               collection.add(3);
               collection.add(4);

// 注释掉这行-VVV-并去掉下面/* */的注释格式化JSON
                return ok(result);
/*
// 为了返回打印良好的JSON,需要将字符串格式化,然后显式地设置响应
// http://www.playframework.com/documentation/2.1.0/JavaResponse
//
//
        ObjectMapper mapper = new ObjectMapper();
        String s="";
        try{
```

```
            s = mapper.defaultPrettyPrintingWriter().writeValueAsString(result);
        }catch(Exception e){}
return ok(s).as("application/json");
*/
    }

}
```

可以在应用所在目录启动内置的服务器：

```
$ play ~run
```

可以删掉下述代码：

```
return ok(result);
```

代之以紧跟其后的代码，刷新浏览器就能看到输出的 JSON 被格式化了。

> **不需要刷新浏览器**
>
> 使用 James Ward（https://github.com/jamesward/play-auto-refresh）开发的 Chrome 插件，浏览器都不需要刷新。每当保存文件时，浏览器会自动刷新。安装 Play Chrome 插件（https://chrome.google.com/webstore/detail/play-framework-tools/dchhggpgbommpcjpogaploblnpldbmen），修改 project/plugins.sbt，为其添加下面一行：
>
> ```
> addSbtPlugin("com.jamesward" %% "play-auto-refresh" % "0.0.3")
> ```

6.2.7　其他轻量级服务器

以上并不是所有构建 API 的选择，Dropwizard（https://dropwizard.github.io/dropwizard/）是一个开发 HTTP/JSON Web 服务的 Java 框架（和 Restlet 功能类似）。它基于其他最佳技术组合（HTTP 使用 Jetty、JSON 使用 Jackson、REST 使用 Jersey）。无容器和最小化嵌入式的服务器对 Java 社区来说还有点新鲜，但来自其他语言的程序员早已把这当成理所当然的了。

6.3　基于JVM的服务器

除了使用 Java 的选项，JVM 也可以作为其他语言的编译目标。每次发布 Java，都会增加对基于 JVM 的语言的支持。事实上，前面介绍的 Play 框架将 Scala 作为其优先使用的语言。如果你对基于 JVM 的语言的表达能力感兴趣，Vert.x（http://vertx.io/）这个 Web 服务器允许使用 JavaScript、CoffeeScript、Ruby、Python、Groovy 和 Java 开发应用。

可以在 Java 应用中调用基于 JVM 的语言，一个项目也可以使用一个 Java 类作为其语言的解释器。下面的代码片段展示了如何创建在 JVM 上运行的简单服务器。

Jython

如果你已经安装好 Python，使用下述命令启动一个 Web 服务器共享当前目录下的文件：

```
$ python -m SimpleHTTPServer 8000
```

Jython 是 Python 在 JVM 上的实现，多花一点儿力气就能完成类似的功能。创建 build.gradle 文件，其中一个任务启动 Jython，调用一个 Python 脚本启动 HTTP 服务器：

```
apply plugin:'java'

repositories{mavenCentral()}
dependencies{compile 'org.python:jython-standalone:2.5.3'}

// 运行Jython服务器的例子
task(startServer, dependsOn: 'classes', type: JavaExec) {
    main = 'org.python.util.jython'
    classpath = sourceSets.main.runtimeClasspath
    args  "http_server.py"
}
```

http_server.py 包含切换到共享文件目录，创建并启动服务器的代码：

```
import SimpleHTTPServer
import SocketServer
import os

os.chdir('root')
print "serving at port 8000"
SocketServer.TCPServer(("", 8000),
SimpleHTTPServer.SimpleHTTPRequestHandler).serve_forever()
```

创建一个名为 root 的文件夹，所有在 root 文件夹里的文件都将通过 `SimpleHTTPServer` 共享。以 .json 为后缀的文件其内容类型是 `application/json`。

模拟服务器的最快方式

有了这个简单的配置，客户端开发者在服务端代码还未完成时就能开始编写客户端代码。通过创建和 API 路径相仿的文件目录，就能模拟 JSON 响应。这些 JSON 响应文件可当作 API 文档，也可以验证服务器端代码的单元测试。

6.4 Web应用服务器

当然，还有完整的 Web 应用服务器方案，用来部署通过 Java、Python（使用 django-jython，参见 http://pythonhosted.org/django-jython/war-deployment.html）或 Ruby（使用 warbler，参

见 https://github.com/jruby/warbler）开发的产品代码。Tomcat 是一个 Servlet 容器（前面提到的 Roo 框架就使用它），因此任何 Web 应用都可以部署成一个 WAR 包在其中运行。包含一组 WAR 包的项目需要被打包成 EAR（或者单独的 WAR 包需要某些 JEE 服务），此时就需要一个完整的 JEE 应用服务器，比如 JBoss。

6.5　如何在开发中使用

客户端和服务器端的分离为客户端开发提供了多种可能，使用快速建立的服务器，让服务器端得以并行工作。最基本的是，只要创建一个 HTTP 服务器，就可以返回硬编码的 JSON。再花点工夫，就可以将 JSON 保存到文件系统中，文件名对应着 API URL。如果需要访问数据库，Roo 包含了方便的数据库集成功能，其他基于 Java 的项目也提供了该技术。服务器端的功能还可进一步扩展，包含不同的数据源、应用缓存和其他一些作为生产服务器的基础功能。如果一个大型应用没有明确地划分客户端和服务器端，就不可能使用这种开发方式。

6.6　小结

本章的例子突出了服务器端开发方式的渐进简化，与之形成对比的是客户端代码不断增长的复杂性。Java 开发者进入了一个与传统打包方式不同的世界，应用服务器的部署需要知道这种替代方案以更好地适应客户端 – 服务器端模型。

不必深入网络细节和低级语言，你也可以创建一个轻量级、专用的 API 或 HTTP 服务器。简单、轻量级的服务器端代码和简化的部署选项源自于对创建高扩展性方案的需求，这在大规模部署时被证明非常实用。这种迁移对程序员的日常工作流程影响深远，这些影响（大多数时候都是积极的）正是下一章要讲解的主题。

第 7 章
快速开发实践

“在一句话中找寻我的家，一个简洁的句子，犹如金属锻造。
并非为了迷惑什么人。并非为了在后代之中获取永恒的声名。
一个未命名的事物需要秩序、韵律、形式，这三个词反对混乱和虚无。”

——切斯拉夫·米沃什

7.1 开发者的生产率

简洁、高效、简单，是当代文化所高度推崇的。或许这是因为和以前相比，现代社会变得相对丰富和复杂了。代码简洁、流程高效的 Web 应用会让最终产品易于维护、易于修改，最终会帮助获取更高的利润。同样，程序员的工作流程和工具也应该高效，避免无谓的复杂。由于选择太多，开发者暂时从编码中抽身，认真思考一下工作流程是否优化、是否高效，就变得特别重要了。

随着 Web 开发方式向客户端 – 服务器端架构的转变，开发流程也面临不断的改变和提高。通过去掉很少用到的配置选项、使用合理的缺省值，减少了大量不必要的工作。简化的工作流程带来更紧凑及时的反馈循环，频繁的反馈让人及时发现和解决产品中的问题。及早发现和修复缺陷提高了生产效率——也让程序员更加开心。更进一步说，它让原本需要大量时间和资源才能开发维护的复杂且高质量的软件，现在有可能花费较少的时间和资源就能做到。

大多草草了事……

众所周知，衡量程序员的生产效率很难。工作时间、每日代码量或者解决的缺陷数目，这样的指标虽然可量化，但不是很有用。由于项目的目标、时间和其预期寿命的不同，很难对它们做比较。没有一个指标能适用于所有项目，足够客观、公正，准确地反映出程序员的工作效率。在实践中，大多数软件开发经理根据开发者给出的估计和实际产出，“艺术地”维护着一张电子表格以准确估计这两者之间的关联度。这样就能理解，为什么通过改进流程会提升个人和组织的效率这个假设会被大家所接受了。

*敏捷开发*最早是以纠正瀑布式开发的缺陷的形式出现的，瀑布式开发倾向于一开始就负重前行，花费大量精力，组织大量活动，项目反而不会取得有意义的进展。这种趋势再发展就是“分析瘫痪”，当敏捷开发被初次引入时，其立即着手开发一个可工作的产品的观点令人耳目一新。遗憾的是，随着时间的推移，这一概念被发展成为一种“做了再说”的方式，早期的很多活动都没有一个定义清晰的目标。因此，如果想要生产提高效率需要，可能这样说有违直觉，需要先*放下工作*。显然，这不是指让你真的放下工作，这是为了项目的质量和长期生产效率，而对暂时的衡量指标和可见进度做出牺牲的能力。

这样做很有挑战性。管理层希望看见手头的项目进展，开发者喜欢编写代码，客户希望看到前者真的着手开始工作了。但是过早开始可能会导致工具和开发方式选型错误（或者不是最理想的）。将不适用的约定或传统引入项目遗害无穷。早期的一点点偏差，都会让项目偏离正确的轨道，随着项目的进展，问题会越积越多。认识到暂时*放下工作*，先做一些分析，会让工作效率更高，产品质量更高，这是需要一些远见的。

预先分析因为采用*伪敏捷开发*（pseudo-agile）而饱受诟病（“伪”字是指这里所讲的并不违背敏捷开发实践）。凡事要三思而后行，这适用于多个层面。开发团队领导者需要做出项目相关的决定，每一个开发者需要了解最佳实践和新兴技术，这对他们的手头工作可能有用。有了正确的工具和方法，会减少完成初始工作的时间和精力，如果做对了，从长远看会开发出一套简单、易于维护的系统。不可将全部精力和注意力放在最紧急的任务上，而应做出根本性调整以高效地工作。在项目中可以采用敏捷开发中欢迎变化的方式，但基础的技术决策和相关开发流程不应有大的改动。

避免孤立地看待生产效率

显而易见，只关注生产效率是不够的。如果绝对孤立地看待生产效率，什么也不做或许是最好的选择！软件质量、可靠性、流畅的交流、功能的正确性、对流程的遵守和约定俗成的规则同样重要。假设其他条件一样，最好能用最少的动作、最少的资源完成一件任务。因此，任何以提高生产效率为名义，对流程的改变，都应该放在一个更广阔的视野下来审视。而且，一个真正高效的流程也同样强调质量、可靠性和其他重要的因素。

没有一个简单的计划或者理论会马上提高生产效率，这多少有点让人失望。真正能提高生产效率的东西都是在实践中，在具体的项目中变干边学，发现和制定的。先把工作完成，日复一日，就看出了流程什么地方需要完善，完善它，这样周而复始。这些积攒下来的知识，会成为在其他项目中提高、优化流程和去除不合理之处的源泉。

理论上说，一个软件项目需要完成若干任务，因此在理想情况下会尽可能以最高效的方式完成。划分成任务后，更容易理解和衡量生产效率。每一项软件开发任务都需要若干人，使用若干台电脑完成。在做项目时，人与计算机，人与人，计算机与计算机之间可进行交互。所有这些，从地理上看可能分布在不同地方，如图 7-1 所示。

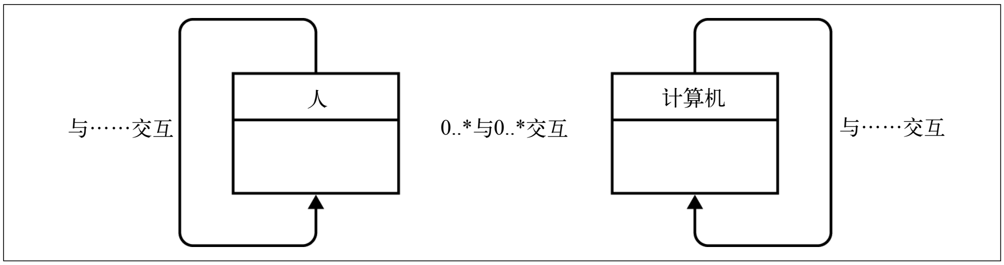


图 7-1：人与计算机的交互

考虑到这一点，就可以在如下交互之间提高生产效率：

- 人与人；
- 计算机与计算机；
- 人与计算机。

人和计算机是完成一项任务所需要的资源，如表 7-1 所示。为了提高生产效率，需要：

- 重新定义任务；
- 提高（给定资源或资源之间交互的）效率；
- 增加资源；
- 花费更多精力（通过加班让单位资源的产出更高）。

表7-1：能提高生产效率的地方

行动	人	计算机
重新定义任务	确定需求、计划、架构和管理	编程语言、软件和编程范式
提高效率	开发技术、最小化干扰	自动化、预处理、压缩、优化、调优
增加资源	开发者、顾问	扩展、增加硬件 / 处理器
花费更多精力	时间管理、调整工作量	并行处理

认识到这些地方的重要性的理由很多。在某个地方没有提高生产效率（比如最近一段时期在计算机领域没有重大的技术创新），会影响到人的生产效率（通过加班或增加人手）。想

一下这些因素和项目的关系能帮助人们有效地行动起来，可能花费相对较小的代价就能带来显著的提升。

这种全盘考虑的方式，能避免过份强调某一方面，去解决所有问题。一个大家都知道的例子是在项目后期，冀希望于增加工作量和人手来赶一个太具有野心的最后期限。另一个例子是自大的开发者陷阱，他们认为不管任务的性质如何，都可以闭门造车，让其自动化。本书强调的重点放在计算机上。

7.2 优化开发者和团队的工作流程

迭代是一种向着最终目标迈进（最终实现目标）、不断循环的过程。项目的每一部分都被分解成任务，经过多次迭代完成，如图 7-2 所示。这种方式适用于软件开发的方方面面，包括需求收集、设计、开发、测试和部署。

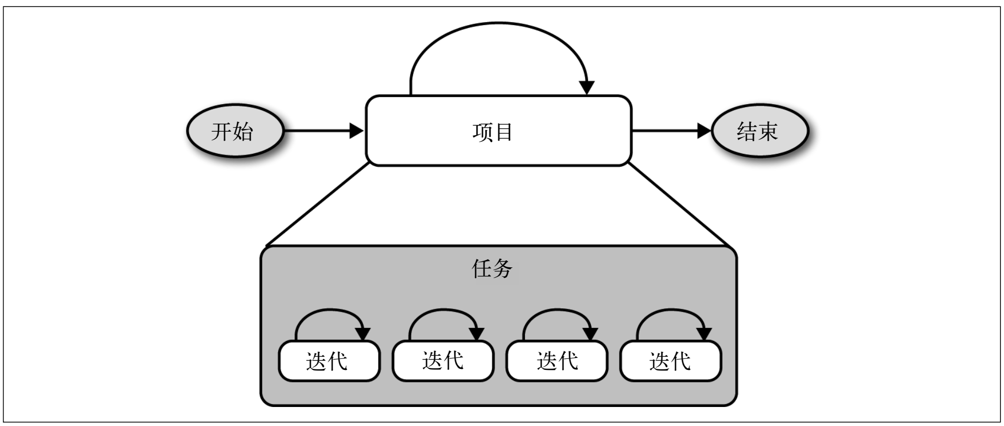


图 7-2：项目迭代

你需要将如下几条谨记在心。

迭代需要以一种可见成果作为终结。迭代完成后的结果需能与上一个迭代的状态，和最终希望的结果做比较。如果结果是不可见的，就说明有问题了。可见的东西才能度量和提高。不可见的东西无法衡量、修复和提高。从这个意义上来说，可将迭代看作一种反馈循环，以行动开始，以可衡量的响应结束。

迭代（或反馈循环）可大可小。开发者对某段代码的细微改动可看作一个小的迭代，而最终发布的一套完整的大型系统则是一个相对较大的迭代。部署后的反馈可基于自动化，也可人工处理。自动化的反馈能提供一个大概的信息，显示系统是否如预期的那样运转正常。但是，自动化代替不了真实用户的反馈。

短周期迭代能得到更多的反馈。更多的反馈意味着看得更多。获取更多反馈的原因很多。

它能快速定位问题（或者机会），更容易在项目早期纠正方向性的错误。循环的周期越短，就越能及时得到反馈，效果就会越好。

从本质上来说，迭代中的每一点进展都会让整个项目取得进展，因为迭代是不断重复进行的。如何发现那些迭代中不断重复的任务，并且改进那些对整个项目影响最大的任务，是一个挑战。

对项目中任何一项任务而言，优化都是值得的。对那些重复多次的任务做优化，好处更多。如有可能，应尽力将任务自动化处理。与其“好好干”，不如减少那些不必要的任务，这听起来看可能有违直觉，但事实却是如此。

这些都显而易见，是简单的常识。但正如那些涉猎软件开发的人所看到的，开发者是习惯的生物。很多人都习惯了某种开发流程或某些工具。这会导致大量不必要的工作，让项目复杂化，最多只能得到次优的结果。

越快越好

Boyd 认为赢得空中格斗的决定性因素不是观察、分析、计划、更好的实施，而是观察、分析、计划和更快的实施。换句话说，主要看一个人能迭代多快。Boyd 认为迭代速度胜过迭代质量。

我将 Boyd 的发现称为 Boyd 迭代定律：在应对复杂分析时，快速迭代几乎总是优于深入分析。

——Roger Sessions，“A Better Path to Enterprise Architectures”
（https://msdn.microsoft.com/en-us/library/aa479371.aspx）

我们中的大多数人都无缘发明什么重大创新，给软件开发流程带来革命性的改变。但是跳出自己的编程文化至少可以开开眼界，有很多现成的改进可供使用，深入研究那些成熟技术，可以让个人获益良多，更不用提对项目的帮助了。让我们通过几个例子来看看。

7.2.1 例子：修复Web应用

现在需要修改一个 JEE Web 应用（由一个包含多个 WAR 的 EAR 构成），该应用使用 Maven 构建，部署在 JBoss 应用服务器上。负责修改的开发者需要对代码做几处改动（在这个过程中或多或少会犯点错误）。解决该问题有一种方法，需要下面几步。首先，修改代码。其次，键入命令启动一个标准的完全构建，然后部署该应用。根据应用的规模、单元测试的数量、构建过程和其他因素，完成第二步也许要花上几分钟。让我们看看怎样做能改善该流程。

首先，有很多 Shell 命令可用，最先想到的是命令行历史（和搜索）。

构建中的每一步是否都需要执行。比如，在 Maven 中，使用 `-DskiptTests` 参数可显著缩短构建时间。

是否需要一次完整的部署。或许不需要一次完整的构建来测试改动，可以热部署改动代码。

改动是否需要一个初始的部署环境。初始的测试也许可以在浏览器里完成（如果改动主要涉及 HTML、CSS 和 JavaScript）。对于服务器端的代码，可以连上远程调试器，在上下文环境中观测相关的对象和变量，可能这就足以发现问题，避免不必要的迭代。Java 代码可以脱离完整的部署环境，用单元测试验证其正确性（这表明了测试驱动开发可以提高生产效率）。

7.2.2 例子：测试集成

这个例子还是围绕上面项目的，但进展顺利。人们一致认为应该加入测试，作为软件开发生命周期的一部分。这可以发生在软件开发的各个阶段和层次。

测试人员和开发人员可以结对测试，验证最初的需求是否和实现匹配，但是人很难保持一致，或者穷尽所有场景。

创建（使用 JUnit）单元测试。单元测试能更全面地测试，但如果不能持续运行，效果就大打折扣了。

将单元测试集成进 Maven 构建中。在一台持续集成服务器上构建，如果有代码变更破坏了构建，开发者会立即受到警告。然而这还是无法说明测试覆盖范围有多广，价值有多大。

单元测试覆盖率报告能给出代码覆盖率的情况。但是测试集中于服务器端，当项目中浏览器端的代码占大头时，这就成了一个问题。

幸运的是，足智多谋的前端开发工程师已经开始使用 Jasmine 来做单元测试。使用一个插件，就能将其集成进 Maven 构建中。而且，JavaScript 开发者可以使用在自己的工作站上安装 Karma，用它来运行客户端代码的单元测试。

随着项目的进展，工作流程也趋于稳定，此时就可以编写功能测试来反映用户的使用体验。使用 Selenium 可以在各种浏览器上运行功能测试。

这种测试场景将关注点放在如何增加对项目状态的反馈，而不是项目的质量上。优化的价值显而易见，缺陷被更快地发现和修复。而且，需求和实现的不一致也能快速被发现。新的开发者能快速融入该项目，因为通过观察和运行测试，他们能相对独立地理解代码。一个单元测试覆盖率高的项目能在大规模重构后很好地活下来。能进行如此重构的信心来源于这些自动化的测试，它们保证了功能的完备。

7.2.3 例子：绿地开发

作为新项目的架构师，你需要负责挑选最适用的工具，建立起系统的初始架构。对于云部署的、高扩展性的 Web 应用，可采用本书描述的客户端 – 服务器端架构。团队采用一些新的工具和流程是允许的，而采用 Java 作为编程语言是必需的，因为这是经过实践的，在内部被证明是能够胜任工作的（这就除去了使用 Rails 和 Grails 等依赖其他 JVM 语言的可能性）。

首先我们想到了 Maven/JEE。虽然 JAX-RS 也适合开发服务器端程序，但 JSF 会诱导开发者使用会话，这不是我们想看到的。

一番调查之后我们发现，可使用 Spring Roo 或 Play 这样的服务器端框架，来减轻构建和部署应用服务器的负担。这里我们选用了 Play，生成服务器端 API，使用一些存储在文件系统之中的示例 JSON 文件向外提供数据。这些模拟的服务会被稍后集成后端的服务替换。

前端项目使用 Yeoman 创建，并可能会用到 JavaScript 框架和相关的 HTML5 项目。键入 `npm search yeomangenerator` 会返回一些备选项，使用它们生成一个或多个客户端项目——各自在独立的目录底下。通过几个小时的调研（将前端项目集成到现有服务上），就能大概了解每个项目的价值，然后选择一个最适用的项目。

最后做一些清理工作收尾，包括提供服务器端和客户端的示范单元测试，文件保存后就能自动运行；使用工具让 Java 和 JavaScript 的文档自动化；将代码提交到 SVN，在持续集成服务器上注册和配置项目；代码构建完成后，服务器生成文档并发送到统一的文档服务器；并创建一份 IDE 模板，包含经过一致同意的代码规范默认选项。

这些前期工作完成后，在开发者实现具体的业务需求前，很多重大决策就已经定下来了。这些决策允许客户端和服务器端进行相对独立（并行）的开发。通过单元测试和具体的例子，开发者为新功能添加测试时可以直接复制，得到快速反馈；也允许发布自动生成的文档和 IDE 模板，鼓励大家相对一致地编码和注释。

这些例子都很主观，随着新技术的出现，毫无疑问会改变和提高。它们的意义在于证明了与其盲目地遵循前人的实践和习惯，对流程做持续改进也是可行的。后面的几节旨在帮助大家进行头脑风暴，发现那些项目中有待提高的地方。

7.3 生产率和软件开发生命周期

需要在软件开发生命周期的各个阶段思考如何提高生产效率。这是因为对于流程一个地方的改进，不一定会带来其他地方生产效率的提升。总的来说，可以将和提高生产效率相关的任务按照收益递减的顺序排出优先级。虽然项目和团队各异，还是可以总结出一些会产生主要影响的因素。总的来说，管理方式和企业文化是最根本的，然后是整体的技术架

构、具体的应用设计和底层编码以及平台选择。对任务有了准确的分析和优先级，就能按照影响生产效率的重要程度优化处理。

7.3.1 管理方式和企业文化

通常来说，从总体上把握由很多人参与的项目能带来最大的收益。虽然这不是本书的重点，但是管理方式、团队活力和工作文化对所要完成的工作影响深远。这些环境因素设定了工作目标、组织和个人的价值观。它们效果显著，常常是最应该重视的地方。统一目标是最大的挑战，尤其对大型组织来说。查理 • 芒格（http://www.ycombinator.com/munger.html），商人和投资家，因与沃伦 • 巴菲特的关系而被人们所熟知，他描述过联邦快递将员工和组织的目标统一起来，从而消除延迟这一挑战。问题的解决之道在于保证所有参与者得到适当的激励：

> 在所有的商业案例中，我最喜欢的有关激励措施的案例是联邦快递。联邦快递的系统核心——创建了产品的信誉——是让所有的飞机在午夜到达同一个地方，然后将包裹在飞机之间分发。如果有延误，整个运营团队就不能把包裹按照所承诺的那样按时送达联邦快递的客户手中。
>
> 他们总是把事情搞砸，从来不能按时完成任务。他们试遍了各种方法——道德说教、威胁，总之各种你能想得到的方法。但是，都不起作用。
>
> 最后，有人想了个办法，降低他们的计时工资，提高计件工资——做完后就能回家。一夜之间，问题解决了。
>
> 保持合适的激励是非常重要的一课。这种解决方案对那时的联邦快递来说还不是很明显。但是现在，对大家来说这都是显而易见的了。
>
> ——查理 • 芒格（http://www.ycombinator.com/munger.html）

还有其他一些“大的方面”需要考虑：那些基本的组织和管理原则依然适用。分工明确、文档集中化管理、使用版本控制，这些看起来都很新鲜，但却常常被忽略。

7.3.2 技术架构

系统的整体架构决定了后续技术的选型和实现。一个在大范围、公开部署的高扩展、基于云的应用和组织内部使用的应用不同，需要更加复杂的配置，后一种场景对错误的容忍度也高。如果成员需要在编码时考虑那些永远也不会发生的场景，或者在项目后期要适应没预料到的架构要求，整个团队的生产效率就会受到拖累。架构选型时应该明确项目的范围。

数据存储方式的选型是另外一个相似的例子。人们对传统的关系型数据库理解相对充分，它提供了引用和事务完整性，开发者认为这是理所当然的。新出现的 NoSQL 方案提供了优化的写操作，以一种限制更小的方式存储数据。每种方案都有自己的优势，但是没有一种锦囊妙计能覆盖所有的情况。出于对数据扩展性的考虑，可能采用 NoSQL 方案。但是如果一开始没有考虑到报表功能，数据就不会以一种利于高效生成报表的方式存储。开发者可能技术出众、效率极高，但是在完成具体的报表任务时，也无法逾越因为采用了错误的数据存储方式而导致的鸿沟。

每一门语言都有自己忠诚的信徒，他们乐此不疲地为各自语言的优点辩论。可以确定的一点是，对于一项给定的任务，某门语言的特性能让其在更短的时间内，更简单地完成任务。比如，如果不需要编译（通过使用 IDE 的自动编译选项或者脚本语言），任务就能用更短的时间完成。像 Java 这样的语言，有大量现成的函数库可用，而其他一些新语言，比如 Scala 和 Groovy，则夸耀和其他语言有根本性的不同，使用较少的代码就能完成同样的任务。像 Ruby 和 Python 这样的脚本语言则有它们自己的工作流，在它们的领域里效率很高，而且也影响到了其他语言所使用的工具和流程。

7.3.3 软件工具

选择编程语言、开发工具和框架是架构师确立项目方向的主要领域。每个程序员在开发项目过程中所具备的能力、所受到的限制都深刻地受到这些决定的影响。技术和与之相关的工作流程都有各自的考虑，生产效率不可避免地会受到这些选择的影响。

在 *Software Tools*（Addison-Wesley Professional 出版，1976）一书中，Brian Kernighan 说过一句名言："编程的精髓在于控制复杂度。"试图降低复杂度的软件工具遍及软件开发生命周期的方方面面：版本控制系统、文档自动化、覆盖率和质量报告、测试工具、事项管理系统和持续集成服务器，这里只是仅举几例。除过这些，每位开发者所掌握的日常工具也能让他和同僚之间在效率上差好几个数量级。

每门编程语言都有与之相关的构建工具。虽然可以混用语言和构建工具，但是基于 Maven 的 Java、Gradle 的 Groovy、SBT 的 Scala、Rake 的 Ruby 都是紧密联系在一起的。每门语言都有开发客户端 – 服务器端 Web 应用的相关框架。Java 有 JEE 和 Spring（借由一款高度自动化的工具 Roo），还有一些较新的框架，比如 Play（同时支持 Scala）。同样，Ruby 有 Rails、Groovy 有 Grails、Python 有 Django。多数框架包含一个嵌入式服务器，以方便开发流程。他们通常包含起始项目，显著地减少了费时的样板代码。选择合适的框架能减少构建时间、省略不必要的完整构建、享受预处理资产管道所带来的好处、方便地和测试进行集成。

IDE 有代码补全、智能搜索、代码导航、重构（将一段代码抽取成一个新方法、在不同种类的文件中重命名变量）、单元测试集成、后台编译等功能。它们是 Java 社区的中流砥柱。

在使用类似 Java 这样的语言工作时，它们价值连城，以致有些开发者觉得程序员如果不使用 IDE 完成一项任务简直难以置信。

使用脚本语言（尤其那些不是为 JVM 而开发的脚本语言）的开发者倾向于使用轻量级的代码编辑器，如果在 *nix 系统的命令行里工作[1]，vi（或者 vim）和 Emacs 和它们内置的一些小工具会为软件开发提供类似（或者更好）的编辑功能。即使你不总是在命令行下工作，精通命令行也是很有必要的，因为很多支持的任务（部署服务器、查看日志、检查服务器性能）都得在命令行下完成。

7.3.4 性能

性能越好的应用，调试起来越快。让迭代周期（反馈循环的长短）最短，修改和验证项目的机会就更多。优化性能差的代码，能为开发者节省出时间干其他的事，而不是在那儿干等着系统响应。一开始选择的 API、算法、数据存储机制和相关的流程会自上而下影响性能。甚至编程语言范型的选择都会影响性能，比如函数式编程（它以避免产生副作用而著称），通常可用来构建高效的缓存机制。它还可以使用非常简洁、易于理解的代码高效遍历数据结构。它简化了流程，使用更少的代码就能完成重构。应用的性能和代码的可读性都会因正确的技术（正确的团队）选择而受益。即使在一个成熟的项目中，也有可优化的地方。比如很多 Web 应用的网络性能都可通过压缩或减少连接次数而受益。这些细节通常在项目的早期被忽视，但在后期可以通过非破坏性的方式来实现。对性能的提升总能带来其他好处，开发者能用节省下来的时间来编码和测试，而无需等待系统响应。

通过更一般的应用设计，RESTful 风格的应用也会提高生产效率。客户端 - 服务器端的架构允许并行开发，调试和维护也更简单，同时简化了很多在其他时候会很复杂的任务。这些益处对开发系统中独立的模块也适用。正确地将项目模块化，会在前期高效地实现项目，而那些扩展性不好的方案可在后期重写。良好的设计会提高日后的生产效率。

7.3.5 测试

早先的测试很随意，边用边测，现在测试已经成为一种正规化的操作了。自动化测试（包括将测试集成进构建过程）是有信心大规模重构系统所必需的，也是自动化部署的基础。自动化测试早先只是断断续续地在运行，频率也不高。现在处理器能力增强了，每次项目构建（或文件保存）时运行测试也是可行的。测试的范围会因项目的成熟度而变化，但在大多数非平凡的 Web 开发中都是必需的。测试已经在 JavaScript 社区树立了牢固的地位，它让人们可以开发出大型的项目，能在各种浏览器和设备上可靠地执行。为了应对基于

注 1：对门外汉来说，星号在程序员的语言里是通配符的意思。*nix 代表“Unix 和类 Unix 系统，比如 Linux”。*nix 还包含苹果公司的 OS X，但是不包括 Windows。可在 Windows 上运行类似 Cygwin 这样的工具，让其看起来像个 Unix 系统。

云的部署，人们开发出新形式的测试，比如 Netflix 公司的 Chaos Monkey（http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html），它通过主动触发失败，帮助开发出可恢复的服务。

合理地执行测试，能方便程序员和计算机之间的交流，也能方便程序在团队之间进行交流。乍一看还不明显，毕竟测试的目的是为了理解和验证软件的可靠性和功能的。一些类型的测试会显著提升大型项目的生产效率。当测试减轻了集成的痛苦、提升了产品质量时，效果就变得显而易见了。遵循行为驱动开发范型的测试，能在代表各领域的项目成员之间搭建一条一致的沟通渠道，通过一种更精妙的方式来提高生产效率。正如 *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*（https://pragprog.com/book/hwcuc/the-cucumber-book）一书所说：

> “当开发者和业务干系人能明白无误地沟通时，软件团队才能以最好的方式工作。一种非常好的方式是合作编写自动化的验收测试来描述任务……当团队能合作编写验收测试时，他们能发展出自己的特殊语言来描述领域内的问题，这可以帮助他们避免误解。”

更好的沟通减少了误解浪费的时间，更好的理解会产生更好的产品。

当测试影响生产效率时该怎么办？

有时候运行测试会成为负担，降低生产效率。编写和维护测试需要时间，运行测试需要时间和资源。和其他开发任务一样，测试的精力和时间本可以用在其他地方。这导致了开发者和其他人在很大程度上减少了测试。

现在将测试集成进 Web 应用已经变得相对容易，很多启动项目都包含了测试相关的配置。为了最大限度发挥测试的价值，需要鼓励形成尊重测试价值、将维护测试当成真正的工作，并且认为这是物有所值的文化。不能说测试会直接提高生产效率，但对大多数拥有较长生命周期的项目来说，测试的价值是不能被低估的。

7.3.6 底层平台

操作系统和安装的基础软件组成了底层部署平台。具有一两台强大处理器的工作站（加上一台额外的显示器以增大屏幕的实际使用面积）就抵得上几年前的一台超级计算机。为 JVM 分配足够的内存、关闭不需要的程序节省资源，这些可能会有一定的价值，但是应用的设计常常才是更重要的因素。在有些情况下，一个更快的文件系统会显著影响构建时间。

是使用集中化的数据库，还是使用开发者各自维护的备份也是一个重要的决定。在开发者

维护的场景里，迁移时可使用 FlyWayDB（http://flywaydb.org/）这样的框架。如果需要远程使用资源，网络就成了问题，和分散在各地的团队一起工作时尤其如此。

7.4　小结

本章有意不提供示例项目。生产效率，或开发时的效率需要后退一步考虑现有的选项是否和手头的项目匹配。项目的各个阶段都有宏观或微观的任务可被简化、自动化，或者更高效地完成。你最好能够“上来透口气”，它能让你充分思考现有的选项，做出最佳的决策。

第8章

API设计

“纸上得来终觉浅，绝知此事要躬行。”

——陆游[1]

解决问题有两种基本方式：由全面的理论开始逐渐深入细节，或者从一个细节开始逐步发展出一套与之对应的理论。“喜欢着手细节的人”通过研究每一个具体问题来解决问题。“喜欢着手理论的人”适合将问题分类到他们自己涉及的理论范畴内。每种解决方式都有其价值，很多问题的解决需要熟练地在两种方式里切换。

哲学范畴

柏拉图在哲学中描述的“共性”是抽象的最高和最根本的“思想的境界”。而他的学生亚里士多德发现了他们的特异性，特别是现实世界的事物。这些的起点来源于不满足的求知欲。在设计分析的时候，起点会影响过程的最终结果。最好的解决方案往往是这两种方式的互补。这些古老的起点与 Elsevier 的著作 *The Handbook of Statistical Analysis and Data Mining Applications*（《统计分析与数据挖掘应用手册》）中提出的现代数据科学技术有关：

> 传统的统计分析在做数据集相关性检索时会遵循关系演绎法，人工智能（例如，专家系统）和机器学习（例如，神经网络和决策树）则会按照归纳法来执行。排除法（或演绎推理）是亚里士多德分析详细数据的过程，计算一系列数据，然后形成出一些基于（或推断）这些数据的数学。归纳是使用数据集中的信息作为“跳板”来得出没有完全包含在输入数据中的通用结论。

注 1：原文为“In theory, there is no difference between theory and practice. But, in practice, there is. —Author Unknown”。

REST 及 Web API 设计实践也是这些起点的代表。罗伊 • T. 菲尔丁以纯粹的抽象术语清晰地表达了 REST。RESTful Web API 受到了 REST 原则的启发，可以解决具体问题，也可以接受不完全符合该理论的实现细节。大部分对 REST 讨论的分歧都可以追溯到各方有意或无意选择的起点。

REST 规范显然为 Web API 的设计者提供了一些立即可以使用的价值。对客户端和服务端进行分区后，HTTP 动词以及标识 Web 资源，已经被明确证明有助于创造各种现实世界的 Web 接口。事实上 HATEOAS（超媒体即应用状态引擎），虽然在理论上引人注目，但是实际上已经难以持续。尤其是在现在的 Web API 实际上都使用了 JSON 作为数据传输格式，使用 HATEOAS 就更加困难了。

8.1 设计的起点

虽然 Web 服务已经以某种形式存在许多年，但是并没有立即转而使用它们作为基础的设计元素。逐步采用的 Ajax，JSON 的出现以及相关轻量的 Web API，首先影响的是已经存在于服务端的 MVC 系统。通过开发者的推动，这些技术的极限已经可以用来创建复杂的单页面应用程序，很明显，服务端驱动的 MVC 方法并没有与这些广泛使用的新技术进行很好的适配。这使得 Web 应用的设计发生了根本的改变，打破了原有实践的延续性。图 8-1 所示的时间线展示了引领 CS 风格 Web 应用的技术发展历程。

切换到客户端 – 服务端开发方法

客户端 – 服务端 Web 开发用到的技术并不新鲜，编程语言变得越来越成熟，但是没有本质上的区别，服务器和浏览器，以及 HTTP 协议从 Web 创建开始就已经存在。然而，直到最近才开始采用一致的设计方法，而这大部分来源于正在开发的各种技术。

起初服务端输出和客户端渲染的那些页面都很简单。20 世纪 90 年代中期，通过服务器端 CGI 和浏览器端 JavaScript 程序才引入了动态内容。因为客户端不够强力，JavaScript 也比较慢，所以更多地是通过服务端来处理和创建动态内容，为了解决这种复杂性在服务端引入了类似模型 – 视图 – 控制器（MVC）这样的设计模式。

MVC 模式依旧是服务器端 Java 开发的主要模式，并且出现在各大标准（例如 JEE）以及框架中（例如 Spring）。后来类似 Ajax 和 JSON 这样的 JavaScript 创新技术立即影响了服务器端的 MVC 开发者，他们开始零零碎碎地使用了这些技术。框架并没有消失，只是他们的设计和使用受到了显著影响。在设计方面，例如 Spring 这类 MVC 框架都采用了“优雅的网址”来反映资源的行为。在使用方面，API 当然可以很好地与传统的 MVC 框架结合。不论用什么框架，今天的许多应用是基于 Web API 而非 MVC 的，实际上这是一种客户端 – 服务端模式。

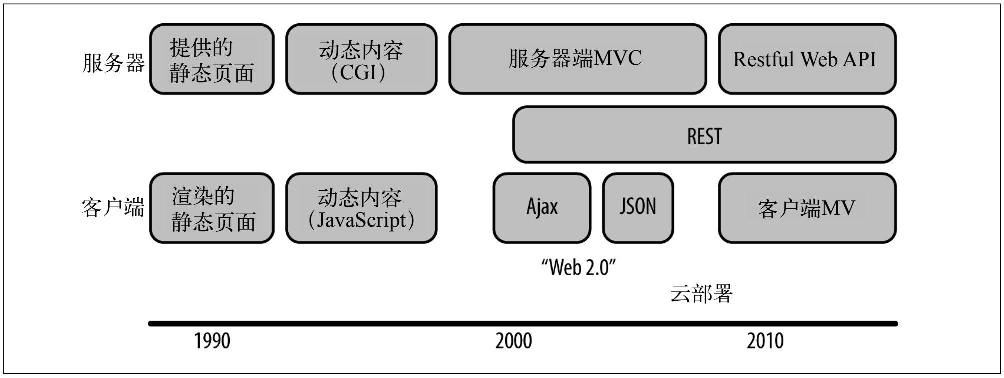


图 8-1：客户端 – 服务端 Web 开发技术历史更迭

客户端 – 服务端模式涉及的一种具体设计方法是：通过开发 RESTful 风格的 Web API 来将数据传输到客户端视图并且避免在服务端做视图的渲染。这样带来的一个问题是：如何以一致、可支持的方式来设计最好的 Web API。这种类型的设计决策常常引发许多观点，但总是会在一个点上达成明确的共识。一般的共识是，受 REST 影响的 Web API 不应该受到有争议或者不切实际的约束的严格限制。

8.2 实用的Web API与REST API

虽然实用性和个人的理解有关，但普遍的共识是认为 Web API 应该易于使用。他们应该易于外部开发者理解、具有一致性、可预测，并且大部分符合 REST。REST 的某些实现提供了易用性设计，虽然 REST 早期的规范并没有特别关注短期的生产力和对开发者的易用性，因为 REST 常常容易理解。

> **菲尔丁如是说……**
>
> “REST 是要做数十年的软件设计：每个细节都是为了延长软件的生命周期并且能够独立迭代。许多限制都是为了对抗短期效率。”
>
> ——罗伊·T. 菲尔丁

那么问题是，REST 在实践中最容易应用的是哪一部分呢？以资源的一致性标识作为名词以及它所遵循的 HTTP 动词是 REST 的基础核心，特别在那些依赖于 CRUD 操作的应用中尤为明显。“优雅的 URL”就是这种标识的一种产物。REST 不涉及具体的性能要求，但其引用的可缓存资源能够有助于系统提高其执行及扩展能力。当然，正如前面章节所描述，对客户端 – 服务端之间差异的了解有极大的价值。REST 一直能有着如此的影响力，因为它的设计理念倾向于促进 Web 设计本身的一致性。工程化良好的应用程序，在部分使用了 REST 的时候，会变得更易理解和扩展。

有些 Web API 的实现与 REST 的理念存在冲突。其中最值得注意的是，使用 JSON 将导致通过媒体类型定义的连接性的缺失。对缺少连接性的媒体类型的使用会直接导致 API 无法自我描述，因此，需要求助于文档。识别 REST 中没有直接提到的一系列动作会更具挑战。在 REST 领域之外，还需要探讨 API 安全性及可维护性等相关话题。

尽管菲尔丁的 Web API 设计并不全面，但确实能应用于其他领域。尽管接口返回内容是否易于阅读并不是 REST 实践特别关注的，但菲尔丁还是特别指出了使用诸如 Java applet 等方案时导致的可见性缺失的问题。这种关注可能意味着在许多情况下，格式化后的 JSON 或许是有好处的。给客户端开发者提供的信息的清晰度比压缩信息节省的那点空间更重要（服务端压缩可以更有效地改善传输大小和性能）

REST 被设想为一个抽象模型。抽象模型经得起时间和空间之外的考验。因此它不会直接考虑系统会随时间改变。出于易用性的考虑，建议所有的改变请尽可能保持向后兼容。API 的版本化正好为这个方向提供了巨大的灵活性。

回头来看，统一的接口其实是 REST 的一种约束。统一的接口几乎完全是自描述的。理想情况下，拥有这些 API 的系统可以不需要文档作为系统切入点。这与 SOAP 之类的协议有着鲜明的对比，它们常常需要引入 Web 服务描述语言（WSDL）来描述 Web 服务的功能。一个 RESTful 风格的系统将提供所有资源的链接，并且不需要任何额外的说明。在实际的应用中，Web API 如果配合上良好的文档则会更加可用。

8.3 指引

与那些规范需要由标准委员会来制订的技术不同，Web API 可以由开发者随意构造，实际上，创建符合更大范围的开发者社区期望的 API 是件很有意义的事情。接下来的小节反映了 REST 在开发轻量级 Web 服务时所带来的显著影响。它们都被推荐为设计简单易懂且可用的 Web API 的指引。

8.3.1 名词即资源，动词即HTTP行为

在 REST 中指定了*资源作为名词*。名词在 RESTful 系统中表示了 URL 路径以及用于操作资源的引用。行为即那些与功能相对应并用于操作资源的动词。在 RESTful 系统中动词特指 HTTP 操作。

资源与对象、实体以及其他设计方法中的表相对应。名词可用 UML 类图中的类建模。它们被建模为实体，之后在 ER 图（Entity Relationship Diagram，ERD）中被实现为关系型数据库中的表。

UML 类图和 ER 图，它们是基于数学的图形符号，但点和边在这些图上还具有额外的意义。

特别关注名词的并非只有 UML 和 ER 图，REST 也同样如此。表 8-1 示意了这些系统之间的对应关系。在 REST 中，名词被用来表示资源；在 UML 类图中它们是类；在 ER 图中，它们是数据库表的实体。在为 REST API 分析和系统建模时可以采用对 UML 类或 ER 图进行分解的方法。在映射到 CRUD 操作的系统中更是如此。

表8-1：模型系统的名词含义

架构	名词表示的实体
面向对象的类层次结构	类
关系数据库	表
REST 接口	资源

在设计包含 CRUD 操作的系统时，需要考虑所有行为对每个资源的适用性。这样可以避免以后因为功能需求被忽略而造成的系统修改。例如，博客引擎的建模可能包含用户、博客文章以及评论等资源。大部分博客系统需要创建、读取和删除这些资源的功能，而更新用户或者修改评论可能就不是必要的。就像是表 8-2 所示，第一列定义的是资源（名词），每一行都定义了行为（动词）。

表8-2：简单的博客接口设计网格

	创建	读取	更新	删除
用户	x	x		x
文章	x	x	x	x
评论	x	x		x

没有相应名词……

REST 模型中资源是架构的基础单位。对 REST 资源进行分析所采用的技术与用于 UML 类图及 ER 图中的类似，能够很好地映射到 CRUD 类型的应用。那些不是基于资源的应用可以被视为一组动词，这样的应用用一系列远程调用来表示更好，这种类型的系统无法简单地通过 RESTful 来表示。

许多情况下，类似这样的系统可以被重构为以资源为基础，但这也并非总是可行。对某些系统的建模可能只要使用抽象，但对有些系统却不够。

8.3.2 请求参数作为修饰符

尽管名词是 URL 中重要的部分，动词会关联到 HTTP 的行为，但这仍然不足以将整件事

情描述清楚。URL 的其他部分会包含额外的信息。正如口语的语法一样，话语中的额外部分将用于确认及澄清名词和动词的意图。

当引用的内容是一系列资源的时候，查询参数就变得十分有用。这些参数可以对整个集合进行过滤并返回其中的子集，它们可能返回选中的一些资源，例如仅返回前 10 条。它们也可以对结果进行排序。

分页操作是过滤操作的一种特例。可以用参数来显式地引用返回结果的子集。请求参数更加适合对集合请求返回的数据集进行限制（你也可以把它当作形容词或者副词看待）。分页的意义超越了请求参数的用法，对它来说更重要的是链接。

GitHub 的 API 提供了一个文档良好的分页参数范例。`page` 参数表明了哪个有效的页面被返回，`per_page` 参数则会将返回结果限制到 100 条：

```
curl https://api.github.com/user/repos?page=2&per_page=100
```

> **HTTP GET 请求及请求主体**
>
> URL 中的请求参数会“丑化”它，不然会“优雅”许多。在某些情况下，对 HTTP GET 而言，请求参数看上去是一个必要的举措，可是如果能够引入一个请求对象，那么将可以更容易地去定义一个分层的数据结构。比如去获取一些数据，它们每一个都是由不同的字段查询拼接而成。罗伊 • T. 菲尔丁及 HTTP 规范看起来支持这样做，尽管许多情况下也可以这么用，但是由于缺少对服务器解析主体的官方要求，所以不建议在系统中长期使用。

8.3.3 Web API版本

REST 并没有要求 Web API 在设计的时候考虑变更及修改，它的约束是为了推广更加轻松可变的系统。接口版本化是菲尔丁论文中未提及的重要考虑因素。在 URL 地址中加上一个版本号可以防止 API 通过干扰客户端的行为对 API 进行修改。如果没有版本控制，那么在服务端改变的时候就需要对客户端做同步更新（这在许多情况下是不可能的，更别说在实际应用中了）。虽然版本标识符理论上不够优雅和简洁，但是却极大地提高了系统的可维护性。这是因为对它们的用法存在很多的意见，并且它们也不具备自描述的能力。版本号在使用的时候的数值范围不是自描述的，所以在 Web API URL 路径中的版本片段需要更容易被辨识出来并且应该有更有效的文档。

对于将版本号放在哪个地方存在大量不同的意见。有些人建议它应该被包含在 URL 路径之中，另外一些人则建议将它作为请求的参数，还有一些人则认为版本信息不应使 URL 变得混乱，他们建议可以将版本信息通过 HTTP 标头来传递。

8.3.4 HTTP标头

版本标识符是唯一一个多用途（https://en.wikipedia.org/wiki/List_of_HTTP_header_fields）的请求和响应头，其他一些非相关信息也可以存储于其中。Twitter 在 rate limit data（https://dev.twitter.com/rest/public/rate-limiting）的报告中提出，当开发者的应用程序即将接近某个临界限制时，可以在头信息中对他们发出警报。ETag（http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html）可以用于控制缓存，其他的头信息可能适用于安全认证和授权。头信息常用于传递次要但必要的信息，虽然它们并不符合某种抽象模型，但它们却有许多实际用途需要我们去考虑。

Accept 和 Content type 这两个头信息可以用于影响服务端决定如何对一个请求进行响应以及提供怎样的响应，除了能够清楚地区分出 XML 和 JSON，还可以用于决定 JSONP 还是 JSON。JSON 内容可以“违反”跨域限制发送到远端服务器的某种能力，从实质上来看，它允许访问 JSON 的 API，其返回的内容是一个包含了 JSON 内容的函数调用。本章后面的项目中将会提供这样的示例。

8.3.5 链接

分页器假定了这种能力，它可以链接到与当前显示子集相关的上一个及下一个资源。有些 API 的设计者会推荐在接口的返回结果中包含上一页及下一页的全部链接，其他人则建议只应包含相关的 ID，从而可以节省空间及去除重复的内容。虽然，在返回内容中提供链接往往是一个不错的决定，并且可以限制对额外文档的需求，严格意义上的 HATEOAS 并不实用，或者说在这点上并不能适用于所有的场景。

8.3.6 响应

拥有一个自描述的 API 是个非常棒的想法。如果在 URL 中指定并反映了资源，并且利用了 HTTP 动词，就可以实现非常好的自描述 API。遵循 HTTP 响应状态码的使用约定（例如，400 表示客户端内容，而 500 表示服务器问题）也可以帮助实现自描述 API。理想的情况是，系统的错误信息可以为问题的触发原因提供快速可操作的描述。但是在大部分系统里，至少需要好几个基础领域的文档。错误码和信息通常有点简洁，因此应该将它们作为文档的索引。理想的情况是，在文档里可以按照系统错误信息对错误进行描述（例如，标识出 PUT/POST/PATCH 的所有字段以及可能引发的错误）。

8.3.7 文档

文档应该能够轻松地定位、搜索和理解。一些 Web API 开发者采用的约定是使用 Web 应用描述语言（Web Application Description Language，WADL），这是一种机器可读的 Web 应用描述，一般为 XML 格式。在这种方式下 API 的检索很简单，但是如果仅仅通过工具

来构建并不足够。开发者可以通过命令行的 Curl 操作来复制提供的示例，示例表明想要构建一个清晰的 API 文档还有很多工作需要做。如果 API 是面向第三方开发者的，那么还有更多的细节需要注意。

RESTful Web API 的文档书写需要很多手动工作，不过，可以通过工具自动生成文档来减少这类工作的负担。一些服务能够基于 Web API 创建可用的 WADL 资源。例如，在运行阶段，Jersey 会创建一个基础的 WADL，你可以通过一个 GET 请求访问 /application.wadl。通过制定选择指令可以增加额外的信息到 WADL 中。如果你使用的是一个无法自动生成 WADL 的服务，可以添加类似 Enunciate（http://enunciate.codehaus.org/）的插件使得你的项目可以自动生成。还有类似 http://apiary.io 的网站，在这类站点上你可以脱离特定项目的上下文来设计和描述 API。

8.3.8　格式约定

最后，遵循简单的格式约定可以使得一个 API 更加清晰明了。在开发初期和后续维护中，开发者都需要经常地查看这些文档。由于 JSON 属于 JavaScript，遵循惯用的 JavaScript 实践比如驼峰命名法是非常有意义的。

另一种简单的做法是格式化的 JSON 响应，这使得人们更易于阅读。针对这一点通常的争论在于格式化的 JSON 增加了响应内容的大小，并且影响性能，不过通过配置开启 JSON 响应的 GZip 压缩可以大幅提升性能。在格式化的响应中增加的空间相对于造成的性能损失来说是值得的，因为开发者可以直接查看返回的响应，而不需要先在 IDE 中进行格式化或者使用类似 jq（http://stedolan.github.io/jq/）的命令行工具。

8.3.9　安全性

REST 没有提供明确的、关于安全性的建议。因为它本来就是基于可以将 API 公布到 Internet 上的要求来设计的。不过，通过 HTTPS 而不是 HTTP 来提供 API 也是一个好主意。对于不打算限制到一个服务器上的 API，一个 JSON 的 API 可以通过 JSONP 或者 CORS 来实现公开。

8.4　项目

下面的项目将演示如何使用 Jersey 来返回 JSON，XML，或者 JSONP 内容。在这个 Hello World 式的应用中只用到了一个资源（greeting）。该项目已经公布在 Github（http://bit.ly/LXhEB8）上。

8.4.1　运行项目

这个项目被设定为从 Maven 中运行一个 Java 类。这个类包含一个主要的方法，在本地的

8080 端口启动一个 HTTP 服务。一个简单的命令可以用于构建和运行这个程序：

```
mvn clean install exec:java
```

8.4.2 服务端代码

服务端代码包含了三个 Java 类。`App.java` 包含运行服务的 main 方法。它创建了一个 `Grizzly`HTTP 服务的实例并且定义了 `/api` 作为 Web 接口环境的根路径。紧接着它添加了一个静态的 HTTP 处理程序，用于响应 HTML 及 JavaScript 代码：

```
package com.saternos.jsonp;

import org.glassfish.jersey.grizzly2.httpserver.GrizzlyHttpServerFactory;
import org.glassfish.jersey.server.ResourceConfig;
import org.glassfish.grizzly.http.server.*;

public class App {

  public static void main(String[] args) throws java.io.IOException{

    HttpServer server = GrizzlyHttpServerFactory.createHttpServer(
      java.net.URI.create("http://localhost:8080/api"),
      new ResourceConfig(GreetingResource.class)
    );

    StaticHttpHandler staticHttpHandler =
    new StaticHttpHandler("src/main/webapp");
    server.getServerConfiguration().addHttpHandler(staticHttpHandler, "/");

    System.in.read();
    server.stop();
  }
}
```

`GreetingBean` 是一个与渲染 XML 响应内容相关的、包含注解的 POJO：

```
package com.saternos.jsonp;
import javax.xml.bind.annotation.*;

@XmlRootElement(name = "greeting")
public class GreetingBean {

  @XmlAttribute
  public String text;

  public GreetingBean() {}

  public GreetingBean(String text) {
    this.text = text;
  }
}
```

`GreetingResource` 提供了一种能力，可以透过服务器返回 `GreetingBean` 包含的数据。Jersey（https://jersey.java.net/）是 JAX-RS 的参考实现，它会将 Web 请求映射到 Java 的方法之上。JAX-RS 支持将注解应用在 Java 对象之上。注解是到 Java 的 1.5 版本才被支持的。它们被使用在框架内适用于约束类和方法的行为，并且有效地减少了完成常见任务所需要的代码量。这些注解有效地提供了 DSL，它可以很明确地映射到 HTTP 下面的功能。

`@GET` 注解表示了视图中的 HTTP 请求动词。`@Path` 描述了环境的 URL 地址，而 `@Produces` 则描述了由 Jersey 在方法中返回 bean 的时候会产生什么样的内容类型。`@QueryParam` 被用于指定 `getGreeting` 方法的请求参数。表 8-3 介绍了典型的注解。

表8-3：部分JAX-RS注解

注解	描述
`@GET`	表示请求资源
`@POST`	在指定 URI 地址创建资源
`@PUT`	在指定 URI 地址创建或更新资源
`@DELETE`	删除资源
`@HEAD`	除了不包含内容主体外，和 GET 一致
`@Path`	资源的相对路径
`@Produces`	用于标识服务器返回的媒体类型
`@Consumes`	用于指明服务器可以接受的媒体类型
`@PathParam`	将 URI 路径参数绑定到方法上
`@QueryParam`	将请求参数绑定到方法上
`@FormParam`	将表单参数绑定到方法上

在 JAX-RS 中还有一些其他的注解，详情请参见 *RESTful Java with JAX-RS*（http://oreil.ly/RestfulJava-JAX-RS，O'Reilly）：

```
package com.saternos.jsonp;

import org.glassfish.jersey.server.JSONP;
import javax.ws.rs.*;

@Path("greeting")

public class GreetingResource {

  @GET
  @Produces({"application/xml", "application/json"})
  public GreetingBean getGreeting() {
    return new GreetingBean("Hello World Local");
  }

  @Path("remote")
  @GET
  @Produces({"application/x-javascript"})
```

```
    @JSONP(queryParam = JSONP.DEFAULT_QUERY)
    public GreetingBean getGreeting(
           @QueryParam(JSONP.DEFAULT_QUERY) String callback
           ){
      return new GreetingBean("Hello World Remote");
    }
  }
```

8.4.3 Curl和jQuery

包含在项目中的客户端代码使用 jQuery（http://jquery.com/）去调用 API 的 URL 路径。jQuery 类库有丰富的 Ajax 功能，并且将与 JavaScript XMLHTTPRequest 核心对象相关的使用复杂度及跨浏览器挑战性隐藏了起来。

在应用程序中的调用可以通过 `Curl` 来重复，如下所示。表 8-4 展示了本例中用到的 Web API URL 路径。

表8-4：应用程序地址

地址	描述
/	Web 应用中 HTML 以及 JavaScript 的目录
/api/greeting	来自 `getGreeting()` 的 JSON 或者 XML 数据
/api/greeting/remote	来自 `getGreeting()` 的 JSON 回调字符串

Curl 可用于返回一个页面的 HTML 代码：

```
curl http://localhost:8080
```

当使用 `Curl` 时，可以指定参数 `-i` 来包含头部信息。HTTP 响应的状态码以及内容类型特别值得关注。例如，如果你指定了一个不存在的服务器地址，它将会返回“找不到页面”的信息：

```
curl http://localhost:8080/api -i
```

默认情况下应用程序会返回 XML 内容：

```
curl http://localhost:8080/api/greeting
```

通过修改 HTTP `Accept` 请求头，则会返回 JSON 的内容：

```
curl http://localhost:8080/api/greeting -H 'Accept: application/json'
```

最后，返回 JSONP 的调用通常包含一个 JavaScript 函数名作为指定的查询参数。这些调用返回一样的由 JavaScript 函数填充的 JSON 调用的结果。由于 JavaScript 文件可以从不同的域下载，因此这些内容能够被返回。如果不这样做，它们会被 JavaScript 的同源策略禁止：

```
curl http://localhost:8080/api/greeting/remote?__callback=myCall
curl http://127.0.0.1:8080/api/greeting/remote?__callback=myCall
```

8.5 实践理论

单纯从理论上来说，REST 是非常理想的标准。它可以作为项目是否遵循它的规划来实现的度量手段。它应该能被研究和理解。其他技术的价值，比如 JSON 也已被证明。基于 JavaScript 的客户端可以很容易地使用 JSON。即便 JSON 缺少被普遍认可的链接机制，也没能阻挡开发者选择它用于数据传输的热情。补充这些 API 和文档是与其他实际上的考虑有所不同的，前者构建了一个完美的理论系统，但它从未落，后者作为实际的解决方案满足了切实的需求。

随着时间的推移，开发者们面对了很多 RESTful Web API 对于实际问题的适应症，这进而改变了架构的方法。这些 API 可以被有各种能力的设备所使用，甚至包括那些由第三方开发者所创建的设备。它比 SOAP 更加轻量，而其他的 Web 服务实现差不多都有着复杂的封包格式和交换模式。它可以用于创建应用，而不会对客户端上的过时数据造成问题。它有效地分发进程，处理有显著计算能力的客户端。当部署在云平台（如 Amazon Web 服务）时，可以简单地增加额外服务器来横向扩展应用。这些以及其他一些特性使得开发者从偶尔实现或使用服务变为使用一致的 RESTful Web API 来开发完整的应用。

第9章

jQuery和Jython

“语言对我们了解世界而言已经非常重要，而不再只是简单的辅助手段。对世间万物而言，词汇也不仅仅是声音标记与交流的次要叠加。它们是社会交流的汇聚成果，以及人类世界组成和表达的基本工具。这是 20 世纪人类对语言的典型观点，它贯穿了人类科学发展的整个历程。”

——罗伊·哈里斯

编程语言已经对人们的生活方式产生了直接而切实的影响，这包括它们的使用者，也包括那些甚至都没有意识到它们的存在的人。有趣的是，程序员通常很少会花时间去做跟语言基本特性有关的工作。在弄明白这些之后，他们会立刻尝试避免重复的无用功，抽象层（泛化）会变得非常受欢迎，近乎成为原始语言的一部分。

JavaScript 相关的 jQuery 库就是这样的一种技术。它已经被广泛使用，大部分 JavaScript 开发者都在使用。有些观点认为它不只是一个库，而更像是一种内部 DSL（内部领域专用语言）。这种观点认为 jQuery 是一个聚焦于 DOM 操作、Ajax 处理和其他通用 JavaScript 任务的小型语言。无论如何，jQuery 使用得如此普遍，以至于你会在网上发现每当有人问“如何在 JavaScript 中使用 X”，答案常以 jQuery 风格给出。

分层抽象让 Java 的底层比该语言自身更加成功。正如 Java 虚拟机（JVM），它处理 Java 编译器产生的字节码，Java 从一开始就特意设计 JVM 为一个抽象层。因其独立的存在，使得将其他非 Java 语言编译到 JVM 上运行成为可能。JVM 是一个精心设计、高性能的程序，是多年研发的成果。就算开发人员对 Java 语言没有兴趣，但仍会从这项基础技术上受益。

本章创建的项目将使用 jQuery 和 Jython（一种基于 JVM 的 Python 实现）来演示一个简单的 C/S Web 应用原型。

9.1 服务端：Jython

Python（https://www.python.org/）最初由 Guido van Rossum（主要作者和该项目的 BDFL——仁慈的独裁者，Benevolent Dictator For Life）发布在 20 世纪 90 年代中期。Python 因其清晰、可读性强、语法规整出名，和 C 语言的风格（比如括号）不同。因为它的一致性和清晰性，它已经在很多教育机构作为编程语言授课的入门语言。Python 通常比 Java 使用更少的代码完成相同的任务。

Jython（http://www.jython.org/）是一个运行在 JVM 上的 Python 实现。这使得能在安装了 Java 的环境上创建清晰简洁的 Python 代码。Jython 同样可以和 Java 对象交互，它引入了一系列嵌入 Jython 或与原生 Java 库连结的能力。

9.1.1 Python Web服务器

作为一种脚本语言，Python 同样也能以 Java 开发人员不熟悉的方式使用。例如，运行一个静态 Web 服务器，它不用编写一行原始代码，就可以在任何目录提供文件服务。你可以简单地切换到一个目录，然后调用如下指令：

```
python -m SimpleHTTPServer
```

9.1.2 Jython Web服务器

使用上面提到的 `SimpleHTTPServer` 创建基于 Python 的 Web 服务器只需要几行代码：

```
import SimpleHTTPServer
import SocketServer
import os

os.chdir('src/main/resources')
httpd = SocketServer.TCPServer(("", 8000),
SimpleHTTPServer.SimpleHTTPRequestHandler)
print "serving at port 8000"
httpd.serve_forever()
```

Jython 可以从命令行调用也可以嵌入 Java 应用中。例如这个从 Java 类中调用的脚本：

```
package com.oreilly.jython;

import java.io.File;
import java.io.IOException;
import org.python.util.PythonInterpreter;
import org.apache.commons.io.FileUtils;
```

```
public class Server
{
    public static void main( String[] args ) throws IOException
    {
        new PythonInterpreter().exec(
                        FileUtils.readFileToString(
              new File("python/http_server.py")
            )
                );
    }
}
```

项目及其依赖可以在 Github（https://github.com/java-javascript/client-server-web-apps/tree/master/Chapter-9-jQuery-and-Jython/jython-json-REST/rest-json）上获得。它可以使用 `mvn clean install` 构建，使用 `mvn exec:java` 运行。

9.1.3 Mock API

静态 Web 服务器有明显的局限性。对于 Web App 开发者最为显著的问题是它不能产生动态数据。然而，可以简单地创建一个包含典型数据的文件去模拟这些 API。例如，一个名字叫 api 的目录里面有一个叫 groups.json 的文件，它可以用 URL http://localhost:8000/api/groups.json 来访问。这个 JSON 文件的内容是 groups（分组）的数组，每个对象都拥有名称（name）、描述（description）和 URL：

```
[
    {
        "name":"duckduckgo",
                "description":"Internet search engine founded by Gabriel Weinberg",
                "url":"http://duckduckgo.com/"
         },
         {
                "name":"angular",
                "description":"Open source JavaScript framework initially created"+
                      " by Adam Abrons and Miško Hevery",
                "url":"http://angularjs.org/"
         },
         {
                "name":"twitter",
                "description":"Online social networking service and microblogging"+
                      " service created by Jack Dorsey",
                "url":"http://twitter.com/"
         },
         {
                "name":"netflix",
                "description":"American provider of on-demand Internet streaming"+
                      "media Marc Randolph and Reed Hastings",
                "url":"http://netflix.com/"
         }
]
```

URL 路径中会映射出相对于根路径的目录，如果指定了相应的扩展名，许多 Web 服务器会返回期望的内容类型。在服务器端配置及开发的过程中，客户端开发人员可以通过 Mock API 进行并行开发。

9.2 客户端：jQuery

自 2006 年发布以来，jQuery 在诸多方面简化了跨浏览器开发。它由 John Resig 创建，意图改变开发者编写 JavaScript 的方式。具体来讲，它追求消除重复工作，令模糊、烦琐的 JavaScript 变得清晰、简洁。

在初次邂逅 jQuery 时，代码中出现的美元符号令人惊讶。这是因为 `$` 代表了 jQuery 对象在库中的命名空间。所以它等价如下：

```
jQuery('a')
$('a')
```

在标准的用法中，JavaScript 在内容体的 `onload` 事件执行后，即页面加载后运行。`load` 事件会在页面全部呈现完成后触发（包括全部资源，如图片）。jQuery 提供了一个可以更早运行的事件，就是在 DOM 就绪之后。注册这个事件的处理程序，具体包括几种不同的语法，例如以下版本：

```
$(document).ready(function() {
//.ready()调用函数
});
```

等同于：

```
$(function() {
//.ready()调用函数
})
```

一般模式是使用 jQuery 查找 DOM 元素并对其加以处理。使用某些类型的模式字符串（CSS 选择器或 XPath）来查找 DOM 元素。处理的方式，可以是简单地读取元素的内容，或涉及它的内容、样式，或通过事件的处理程序联系行为。jQuery 同样提供了一个一致的 Ajax 处理接口，同样也设计了插件机制来扩展自身。

9.2.1 DOM遍历和操作

多数对 JavaScript 的批评主要集中在它对浏览器 DOM 的交互上。jQuery 简化了 HTML 与 JavaScript 的交互，为 DOM 选择、遍历、操作提供了优雅的接口。这里最常用的是 CSS1-3 的 CSS 选择器，以及一些 jQuery 的特殊特性。CSS 选择器是一种包含模式匹配的字符串，通过标签名或者一系列复杂的匹配条件可以轻易地匹配到任何一个 HTML 元素。如果给定的元素能够匹配到某个模式的所有条件，这个选择器就可以将这个元素匹配出

来。表 9-1 给出了一些 jQuery 的例子。

表9-1：jQuery的例子

选择器	描述
`$('$div')`	全部的 `div`
`$('#myElement')`	ID 为 `myElement` 的元素
`$('.myClass')`	全部 `Class` 为 `myClass` 的元素
`$('div#myDiv')`	ID 为 `myDiv` 的元素
`$('ul.myListClass li')`	`Class` 为 `ul.myListClass` 的 `ul` 下面的列表项
`$('ul.projects li:first')`	`Class` 为 `projects` 的 `ul` 中的第一个列表项

表 9-2 中展示了其他属性同样也可以被访问，哪怕只知道部分的值也可以将它们找出来。

表9-2：jQuery的部分值

选择器	描述
`$('input[name*="formInput"]')`	`name` 有 `formInput` 子串的 `Input` 元素
`$('input[name^="formInput"]')`	`name` 开始于 `formInput` 的 `Input` 元素

有特殊意义的字符必须使用两个反斜杠来转义。例如，一个 ID 为 `myForm:username` 的文本输入字段，下面的选择器用来识别它：

```
$('input#myForm\\:username')
```

CSS 选择器非常有表现力，但有时得到的对象数组结果还需要做进一步的处理。许多 jQuery 方法会返回一个 jQuery 对象，你可以继续调用其他的方法。比如这个例子，从一个 DOM 子树寻找一个特殊元素：

```
$('div.projects').find('.project1')
```

9.2.2 实用函数

jQuery 同时也提供了一些用来处理类似元素集合的实用函数（http://api.jquery.com/category/utilities/），参见表 9-3。underscore.js 这样的类库与 jQuery 有所重叠，但其提供了更多处理列表和对象的能力，随着时间的推移 JavaScript 也增加了类似的方法。因为要兼容古老的浏览器，所以 jQuery 的方法在短期内应该会被继续使用。

表9-3：实用函数

	jQuery	Underscore	JavaScript 1.6
迭代	`each`	`each`	`forEach`
变换	`map`	`map`	`map`
过滤	`grep`	`filter`、`where`	
索引查找	`inArray`	`indexOf`、`lastIndexOf`	

在一个对象被定位后通常会要做些琐碎的事情，函数的链式调用会让操作元素变得特别方便：

```
$('div.items').find('.item1').text('Hi World').css('background-color', 'blue')
```

通过修改 DOM 中的特定元素，可以开发出响应用户动作的动态交互用户界面。

“查找一个元素”然后“做点什么”这种模式非常简单、强大、直观。令人欣慰的是，只要在一个引入了 jQuery 的页面上打开浏览器控制台，然后开始写入选择器就可以返回对象。当定位到你要找的元素之后，可以尝试对它的文字或样式进行修改。如果你希望在没有引入 jQuery 的页面也这样做，你可以按照下面的方式来做：

```
var script= document.createElement('script');
script.type= 'text/javascript';
script.src= 'http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js';
document.head.appendChild(script);
```

9.2.3　效果

jQuery 拥有便捷的方式可以通过改变 CSS 来显示或隐藏元素。它还包含一些与动画相关的方法，例如渐隐渐现和滑动效果：

```
$('form#myForm').hide()
$('form#myForm').show()
$('form#myForm').slideUp('slow')
$('form#myForm').slideDown('slow')
$('form#myForm').fadeIn('slow')
$('form#myForm').fadeOut('slow')
```

9.2.4　事件处理

jQuery 选择器可以用来向指定的元素绑定事件处理器。随着时间的推移，实际使用的语法已经有所变化。像 `bind`、`live` 和 `delegate` 之类的方法已经被 `on` 和 `off` 所取代。动态接口的增多带来的一个挑战是：你可能需要定义事件处理器，用来处理用户交互过程中生成的元素。解决方案就是：在更高层的 DOM 上绑定这个事件。当事件被触发时，尽管这个事件并没有直接关联到这个元素上，但它仍然会向上传递，直到匹配到合适的选择器，然后进行处理：

```
$(document).on('click','.myClass', function(){
        console.log('点人家做啥啦');
});
```

9.2.5　Ajax

jQuery 将 XMLHttpRequest 浏览器对象封装得更简单易用，在不同浏览器下表现一致。

`jQuery.ajax` 是一个通用的 Ajax 请求器，可以简单方便地应用于 HTTP GET 和 POST 方法。由于在 Ajax 应用中 JSON 通讯的普及，它提供了 `getJSON` 方法，同时还提供了通过 JSONP 来创建调用的能力。

jQuery 是一个出色而且简洁的库。本章着重于概括地介绍它的大部分设计理念。至于如何进一步用 jQuery 来解决特定问题，可以参考 Cody Lindley 的 *jQuery Cookbook*（http://oreil.ly/jquery-ckbk，O'Reilly）等书。

9.3　jQuery和更高级的抽象

jQuery 极大地简化了跨浏览器开发，使得带有重 Ajax 交互、事件处理和 DOM 操作的 Web 应用的开发变得更加简单。它的受欢迎度（http://trends.builtwith.com/javascript/jQuery）表明它还还会在很多年内继续流行且持续影响。但是随着 Web app 应用规模的增大，为了降低复杂度，新的方法出现了。更大尺度的设计模式（MVC）和编程范式（函数式编程）成为了 jQuery 中部分功能的替代及补充。

举例来说，给变量赋值这一简单的场景，这种基本任务也会随着程序中（互相依赖的）变量的增长而变得繁重。而在面向对象的编程中，一个对象可以封装一组变量，用来表示这个对象的状态。可以给这个对象定义方法来访问和改变它的状态。

在 JavaScript 社区，流行使用其他方法而不是（传统意义上的）面向对象来解决变量同步的问题：在模型和视图组件之间进行双向数据绑定（AngularJS）；以及随着时间的推移定义表示值的数据类型的响应式编程（而不只是关心特定时刻的变量的值）。

函数响应式编程

函数响应式编程（Functional Reactive Programming，FRP）是一种最近广受关注的 GUI 设计的声明方式。它的亮点在于许多通过 jQuery 直接声明的构造（事件处理器、回调和 DOM 操作）并非直接完成的。

jQuery 也有它的局限性，所以在客户端模板、模块化代码、回调管理上有更加完善的独立项目如雨后春笋一般出现，它们设法简化直接操作 DOM 的复杂度，多数都可以和 jQuery 一起使用。jQuery 成功地改善了浏览器不兼容的问题，为 DOM 操作提供了统一接口，使它成为了 JavaScript 开发中不可缺少的一部分。

9.4　项目

在本章前面介绍过的基于 Jython 的 HTTP 服务器，会简单地对特定目录内文件的请求进行响应。这用来响应 HTML 和 JavaScript 文件已经足够了。尽管服务器本身提供的功能很

少，我们仍然有办法通过调用第三方 API 来扩展服务器层。一个可以公开使用且不需要特别设置的 API，或者密钥是 GitHub API 的 API。其所描述的应用可以完成从在 GitHub 中查找数据到展现项目成员的任何事情（如图 9-1 所示）。

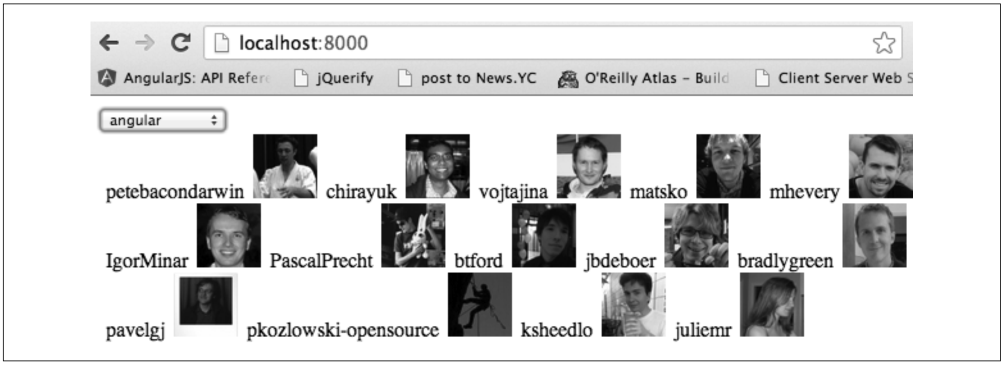


图 9-1：GitHub 组

9.4.1　基础HTML

这个应用可以由一个包含内嵌 CSS 的简单 HTML 文件从头开始“构建”出来。在以这种方式开发时，把所有的代码都放在同一个视图里会更简单，不过在生产项目中，样式表最好是外置的（和之前无侵入性 JavaScript 相关的讨论吻合；关于“不可见 JavaScript”，请参见 http://en.wikipedia.org/wiki/Unobtrusive_JavaScript）。

```
<html>
<head>
        <title>索引</title>
        <style type="text/css" media="screen">
                img {width: 50; height: 50;}
                span {padding: 7px;}
        </style>
</head>
<body>
        <select id="selected_group">
      <option>选择组</option>
    </select>
        <div id="images"></div>
</body>
</html>
```

9.4.2　JavaScript和jQuery

在闭合的 </style> 标签之后添加来自 Google CDN 的 jQuery 引用：

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.8.3/jquery.min.js">
</script>
```

下拉菜单的选项列表，会以 JSON 文件的形式被提前加载。

以下的 jQuery 代码会载入这个 JSON 文件：调用 `$.getJSON`，然后循环访问每个返回的记录并将选项添加到这个选择元素上。每当一个选项被选中时，函数 `getGroup()` 就会被调用：

```
<script>

$.getJSON("/api/groups.json",
        function(data) {
                        $.each(data, function(i,item){
                                $("<option>"+item.name+"</option>").
                appendTo("#selected_group");
                        });
                }).error(function(){ console.log("错误");});

$(document).ready(function() {
        $('#selected_group').bind('change',
        function (){
           getGroup();
        });
});
</script>
```

函数 `getGroup()` 清除之前的显示，然后发送一个请求到 GitHub 来获取选中组织的数据。随后每个组织成员的名字和头像便会显示出来。

```
function getGroup(){
        $("#images").empty();
                $.getJSON("https://api.github.com/orgs/" +
            $('#selected_group').val() + "/members",
                        function(data) {
                                $.each(data,
                   function(i,item){
                                        $("<span>" +
                        item.login +
                      "</span><img/>").
                        attr("src", item.avatar_url).
                        appendTo("#images");
                               });
                }).error(function(){ console.log("错误"); });
        }
```

这个例子展示了使用 jQuery 调用本地或远程 API 来显示结果是非常容易的。整个客户端仅仅包含了不到 40 行的 HTML、CSS 和 JavaScript 代码。jQuery 提供了良好的跨浏览器支持，显然，这也就是它能够这么快速并且广泛地得到采用的原因。

这就是说，这个项目不会在不同的设备上有特别好或者特别差的响应。含有 HTML 片段的字符串可能会让你觉得不适，并且希望得到某种用模板来解决的方案。你也可能会觉得使用嵌套函数有点外行，可能会有一种方式以更自然的函数语法来调用它。你不是第一个

有这种反应的。JavaScript 世界已经对此开发了相应的项目和库，我们会在稍后的章节讨论到。

9.5 小结

以客户端 – 服务器的方式进行 Web 开发只需要几分钟的初始设置。本章中包含的服务器项目仅仅服务于静态资源。静态 HTML 文件和存储在 JSON 文件里的模拟 API 调用可以创建在文件系统上，不需要编译就能更新。创建完成后，jQuery 就可以被用来创建本地或远程 API 调用并显示结果。这种性质的项目并不需要很深入的 Java 或者 Python 知识，也只有很少的 JavaScript 知识需要掌握。结论就是：一个简单的动态 Web 应用可以被不同种类的浏览器显示。

第10章

JRuby和Angular

“我们生活的世界美丽动人且秩序井然，尽管它有时候看起来混乱无常。”

——M. C. 埃舍尔

自古以来，世间万物都通过排序和筛选来组织。古代离合诗[1]的每一行都以字母开头。这既需要筛选（选择第一个字母）也需要排序（按字母顺序排序）。图 10-1 所示为筛法（Sieve of Eratosthenes，又称埃拉托斯特尼筛法，http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes），我们可以把它直观地写出来（http://clientserverweb.com/sieve-of-eratosthenes.html），只需将从 2 到最大值的所有整数排序，然后筛选掉其中为质数倍数的合数（非质数）。

2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31	32	33	34
35	36	37	38	39	40	41	42	43	44	45
46	47	48	49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64	65	66	67
68	69	70	71	72	73	74	75	76	77	78
79	80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99	100

图 10-1：筛法

在网上搜索筛选和排序会返回许多关于在表格程序中操作数据的结果。本章的项目将展示

注 1：离合诗，每行的某些字母组合起来能构成词的一种诗体。——译者注

JRuby 和 Angular 如何使用相对较少的代码，对一个包含谷歌财经股市数据的 HTML 表格进行筛选和排序。

10.1 服务器端：JRuby和Sinatra

使用 Ruby 和一个名为 Sinatra 的微型框架可以创建简单、动态的网络 API。Sinatra 实质上就是一个基于 Ruby 的 HTTP 包装器。

Ruby 是由松本行弘（Yukihiro“Matz”Matsumoto）开发的，他融合了他喜欢的多种语言（Perl、Smalltalk、Eiffel、Ada 和 Lisp），开发了这门能平衡函数式和命令式方法的新语言。尽管松本行弘在 1995 年就发布了这种语言，并且还在 2001 年为其写了一本名为 *Ruby in a Nutshell* 的书，但是直到数年后 Ruby on Rails（或者简称为 Rails）开始流行起来它才广为人知。用松本行弘的话来说，这门语言“表面看起来很简单，但内部却十分复杂，就像我们人体的构造一样”。Matz 最近又与人合著了一本关于 Ruby 的书（http://oreil.ly/ruby-prog-lang）来深入探讨这门奇妙语言的细节。

Rails（http://rubyonrails.org）是 Ruby 的 MVC 网络框架，它让 Ruby 第一次得到了许多开发者的注意。David Heinemeier Hansson 从 Basecamp（他在 37signals 公司开发的一个项目管理工具，参见 https://basecamp.com。37signals 公司网址为 http://37signals.com/）中提取了这个框架并于 2004 年发布。在许多语言和框架中，不写任何代码就不会有任何行为。而在 Ruby（包括 Rails 和其他 Ruby 项目）中，已经包含默认行为，即使不写任何代码。Rails 的理念表明了这一原则：约定高于配置。这大大减少了一个项目启动和运行所需的步骤。这个原则在用 Ruby 编写的其他网络框架中也十分明显，特别是 Sinatra（http://www.sinatrarb.com）。Sinatra 比 Rails 小得多，不需要更大的框架中的附属项目，非常适合创建流线型应用。

10.1.1 工作流

一种使用 JRuby（或者其他 JVM 语言）的方法是从 Java 的角度将其纳入，包含该语言实现的模块可以作为一个依赖引入到项目中：

```
<dependency>
    <groupId>org.jruby</groupId>
        <artifactId>jruby-complete</artifactId>
        <version>1.6.3</version>
        <type>jar</type>
        <scope>compile</scope>
</dependency>
```

在 Web 应用下，如果你已经安装了 Java 应用服务器，这是有效的。Warbler（https://github.com/jruby/warbler）可以用来将 Rack 应用捆绑到 WAR 文件上，有一些有趣的试验，

比如 Rack Servlet from Square Engineering（http://corner.squareup.com/2013/07/rack-servlet.html），将一个基于 Ruby 的服务器端小应用嵌入到一个现有的 WAR 中。

以这种方式使用过 Ruby 的 Java 开发者将会从关注语法差异中获益，但会错过一些可以帮助语言获得成功的工具和工作流。这些差异首先在于用来初始设置语言和相关软件包的工具。

10.1.2 交互式Ruby shell

交互式 Ruby shell（IRB）是可以执行 Ruby 命令的 shell。它相当于一个操作系统的 shell，它计算的任何表达式都能立即得到结果。在很多其他语言中，都有这种读取 – 求值 – 输出循环（Read-Eval-Print Loop，REPL）的程序。当尝试和学习一种新语法，或不确定使用什么算法和数据结构时，通常的编辑 / 编译 / 运行 / 调试过程会非常烦琐。Java 没有任何类似的工具。当调试器暂停时执行一些代码与此类似，Eclipse 集成开发环境中有一个名为 Scrapbook Page 的功能也是类似的。在学习 Ruby 时，IRB 是值得研究的；当使用 JRuby 时，它也能用来获取 Java 的类。

关于如何使用 IRB 来访问 Java JAR 的示例，可参见附录 A。

10.1.3 Ruby版本管理器

不同于在 Java 项目中将 JRuby 作为一个依赖来获取，JRuby 环境可以使用 Ruby 版本管理器（Ruby Version Manager，RVM，参见 https://rvm.io/）进行设置。RVM 不仅限于 JRuby，在其他 Ruby 项目中也能使用。RVM 支持许多 Ruby 环境的部署。每个 RVM 都是自足的，包含了一个特定版本的 Ruby，并且还和所需的 gem 关联。RVM 让项目建立变得容易，因为它包含一组依赖特定版本 Ruby 的 gem 组，并让这些项目独立于使用其他 Ruby 版本的项目。它是交互性的，并且对安装过程中的额外配置或故障诊断给出了智能意见。如果你正在参与多个 Ruby 项目，它可以让你在一个给定环境中进行开发，并能通过一个命令切换到另一个环境。

Ruby/JRuby/RVM/ 依赖管理

Ruby 有多种不同的使用方式，这也是混乱的根源。

Ruby 是一种独立的编程语言。JRuby 是一个能在 Java 虚拟机上运行的 Ruby 版本。RVM 提供了对多个不同 Ruby 版本进行管理的机制，其中也包括 JRuby。RVM 环境包含了安装语言和相关联的 Ruby 包，并且在项目外的本地机器上维护。RVM 提供了一种从 Ruby 角度操作 Ruby 项目的方式，其中也包括 JRuby。

JRuby 也可以作为一个标准的模块依赖被 Maven 或其他构建工具引入。这种情况不需要使用 RVM，而是从 Java 的角度对待 JRuby。

J 可以使用如下方式检测 RVM 当前使用了哪个版本的 Ruby：

```
$ which rvm-auto-ruby
/Users/cas/.rvm/bin/rvm-auto-ruby

# 通常不能直接调用……
$rvm-auto-ruby --version

# Ruby的版本和上条命令的版本相同
$ruby --version
```

列出已经安装的 Ruby 解释器和当前正在使用的版本：

```
rvm ls
```

为切换到另一个环境，可以调用 `rvm use` 命令并跟着 Ruby 解释器名字的一部分：

```
rvm use 2.0.0
```

其他 RVM 函数

除了交互式地维护 Ruby 版本，RVM 还有很多其他功能。通过它，你可以创建一个特定于项目的 Ruby 环境 gemsets（https://rvm.io/gemsets/basics），使其独立于其他的 Ruby 项目。它可以在任何命令行中被用来执行特定 Ruby 版本和 gemset 的脚本。它能够提供的不仅仅是简单的诊断建议，举个例子，有时候通过运行下面的 RVM 命令可以很轻松地解决 SSL 证书问题：

```
rvm osx-ssl-certs update all
```

10.1.4 包

Ruby 的包叫作 gem，通过 `gem` 实用工具进行维护。gem 可以用不同的版本来发布，一个项目可以使用一组相对特定的依赖关系来开发。一组 gem 可以关联一个单独的 RVM 环境，允许多个使用不同 Ruby 版本和不同 gem 组的 Ruby 项目同时开发。为了把一个给定的项目部署到其他机器上，特定的依赖组可以使用打包 gem（http://bundler.io）声明式的维护。打包的依赖列在 Gemfile 中。

使用 RVM，可以创建一个新的 Ruby 环境。之后可以独立安装一组 gem，并且调试环境，直到所有的正确依赖和版本都是可用的。然后，将当前环境使用的所有版本信息补充到打包的 Gemfile 中。下面的 bash 命令可以用来创建一个新的 Gemfile，其中添加了一条注释说明当前使用的 Ruby 版本，并包括了符合当前环境的一组 gem。大部分工作都在最后一条命令中完成，它列出了当前环境可用的所有 gem，格式化了列表，移除了 `rvm` 和 `bundle` 的引用，并使用这个列表来创建有统一 gem 入口的 Gemfile。

```
#!/bin/bash
bundle init

echo "# Ruby Version: `ruby --version`">>Gemfile

gem list --local |
grep -v '\*\*'|
sed 's/[)(,]//g' |
egrep -v 'rvm|bundle'|
awk '{print "gem \""$1"\",\""$2"\""}' >>Gemfile
```

生成的 Gemfile 可以分发到其他需要相同环境的机器上（比如其他的开发机或者独立部署环境），否则，你就需要安装 Ruby 或者 RVM，然后再烦琐地手动安装 gem 进行设置。

10.1.5 Sinatra

正如 Sinatra 网站（http://www.sinatrarb.com）上所描述的，Sinatra 是一个“用最小成本快速创建基于 Ruby 的 Web 应用”的 DSL。本质上，它是一个运行在 Rack（一个 Ruby Web 服务器和 Web 框架的通用接口）上使用了 Ruby 可访问的接口的 HTTP 包装器。Sinatra 应用由路由（每个 HTTP 方法对应一个 URL 匹配模式）组成。一个路由关联一个块，可用于实现相应请求（通过执行 Ruby 代码，渲染一个模板，等等）。

默认条件下，Sinatra 也支持 public 目录下的静态文件服务器（如图 10-2 所示）。这使得它成为一种理想的工具，可以创建一组在 HTML、CSS、JavaScript 应用后台的基于 Ruby 的网络 API。它绝不局限于这种方法。一个标准的服务器端 MVC 方法采用了 view 目录（默认条件下）中基于 Ruby 的模板（ERB）。

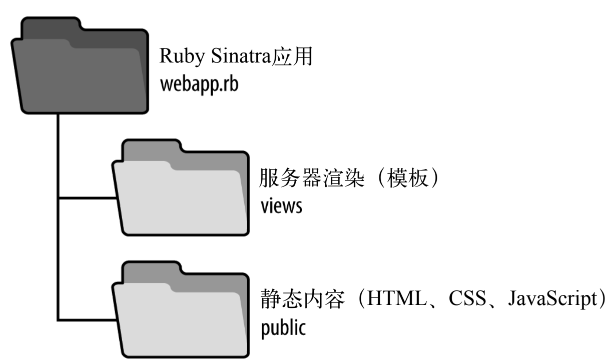


图 10-2：Sinatra 的默认目录结构

Sinatra 非常小，因此有时候被当作一个微型框架。和 Ruby 库中其他的最好框架一样，它能把自己封装起来，只露出清晰的接口供你引用。

或者更准确地说，作为一门领域专用语言（http://martinfowler.com/books/dsl.html），它提高

了程序员的生产力和沟通效率，因为它和它所代表的领域（HTTP Web 应用）紧密结合。

Sinatra 适合于从小应用（按经典方式进行设计）扩展到一种被认为更适合于生产部署（模块化的编写风格）的形式。因此可行的方法是从一个小的 Sinatra 应用开始，然后将其逐渐打造成最终的应用。因为其简单且透明的特性，Sinatra 也可用于原型设计，产出一个基本可以作为正式文档来实现独立系统的应用。

Sinatra 是一个相当简单的框架，有很棒的在线文档（参见 http://www.sinatrarb.com/documentation.html 和 http://www.sinatrarb.com/intro.html），并且已经得到广泛的关注，甚至都有专门为它而写的书（http://oreil.ly/Sinatra-UR）。Sinatra 本身就值得我们学习，并且它还促使其他语言中产生了很多类似的框架。

10.1.6　JSON处理

JSON gem 非常直观，它能进行 Ruby 对象和 JSON 字符串的相互转换。通过 JSON 类型和 Ruby 类型之间的直观映射，JSON 数据处理变得非常轻松。考虑如下的 IRB 情形：

```
>require 'json'
 => true

> o = {:A=>[1,2,3], :B=>{:C=>:D}}
 => {:A=>[1, 2, 3], :B=>{:C=>:D}}

> o.class
 => Hash

> s=JSON.pretty_generate(o)
 => ...

> puts s
{
  "A": [
    1,
    2,
    3
  ],
  "B": {
    "C": "D"
  }
}

s.class
 => String

o2=JSON.parse(s)
 => {"A"=>[1, 2, 3], "B"=>{"C"=>"D"}}

o2.class
 => Hash
```

Ruby 爱好者可能会注意到这里有一些微妙之处（比如，Ruby 的符号和字符串都被转换成了 JSON 字符串）。但总体来说，这个例子展示出使用这个包来解析和生成 JSON 字符串是非常简单且直观的。

10.2 客户端：AngularJS

AngularJS（通常简称为 Angular，http://angularjs.org/）是 Miško Hevery 和 Adam Abrons 于 2009 年提出的 MV* JavaScript 框架。直到 2014 年，一个包括 Igor Minár 和 Vojta Jína 在内的 Google 技术团队负责维护这个项目。Angular 读取其相关 HTML 结构中的 DOM 元素，传递自定义元素和属性（指令），并且将数据绑定到页面对应的模型。

Angular 旨在提供一套完整的 Web 应用开发方案。它利用了 HTML 的声明性并对它的行为进行了增强。这个框架十分复杂，以至于需要一本完整的书（http://shop.oreilly.com/product/0636920028055.do）来进行深入介绍。本书并不会详细讲解这个框架，但会演示如何快速开启并运行。为了更加有效地使用 Angular，我们需要了解一些相关概念。

10.2.1 模型

如果你开发过服务器端 MVC，那么可能会将模型看成一个包含了属性并与关系型数据库相关联的类。Java 开发者通常使用 Hibernate 和 iBatis，而 Rails 中的 ActiveRecord 也可以用来实现相同功能。还有一些类似 Backbone 的 JavaScript 框架，它们包含只能被框架自身识别的模型对象。

在这方面，Angular 是完全不同的。不像其他的 JavaScript 框架那样有特殊的模型对象，Angular 模型是个简单的 JavaScript 对象。Angular 中的*模型*仅仅是数据。因为人们很少关注 Angular 中的模型，正如 Angular 常见问题（https://docs.angularjs.org/misc/faq）中所提到的，一些人更多地将它视为一种模板解决方案。然而，框架的其他特性，包括双向数据绑定，令 Angular 不仅仅是一个模板系统。

10.2.2 视图

*视图*是模型通过 HTML 模板的展现。当模型发生变化时，Angular 的双向数据绑定处于激活状态，以便渲染视图中需要更新的部分。*表达式*是被花括号包围的类似 JavaScript 的代码片段，例如 `{{ _expression_ }}`。

我们可以通过*指令*可以有效地扩展 HTML 的功能。在 DOM 编译过程中，指令会执行并完成各种任务，包括 DOM 操作（如显示或隐藏元素、循环和创建新的元素，等等）以及其他任务。Angular 自带一系列指令，程序员也可以添加额外的指令来实现类似独立 Web 组件的服务。

在 JavaScript 中，普遍存在的问题之一就是全局变量。Angular 减轻了对全局命名空间的污染。Scope 允许模板、对象和控制器共同工作。它们被嵌套在 DOM 结构对应的分层结构中。Scope 会检测模型的变化并为表达式提供执行上下文。

10.2.3　控制器

控制器负责构建模型并将它发布到视图。控制器的相关代码包含在一个 JavaScript 函数中，`ngController` 会将控制器函数与视图绑定。

Angular Seed 项目（https://github.com/angular/angular-seed）提供了一个关于请求是如何被路由到不同控制器的例子。`$routeProvider` 服务通过一个函数将 URL hash 之后的部分和对应的模板、控制器进行关联。

10.2.4　服务

Angular 中存在各式各样的服务。`$parse` 服务解析表达式。所有 Angular 应用中的 `injector` 都会用于定位服务。指令也是如此，开发者可以封装自己的逻辑来实现个性化服务。

10.3　比较jQuery和Angular

人们会很自然地将 jQuery 和 Angular 进行比较。两者的流行程度及影响程度都很高。jQuery 显然是公认的业界领袖，而 Angular 却能够解决一些在单独使用 jQuery 时会遇到的问题。粗略地看下 Angular 文档，我们会发现 Angular 兼容 jQuery，并且有些网站会发布同时使用 Angular 和 jQuery 的例子。虽然它们都可以用来快速开发大型 Web 应用，但是还有许多会影响到设计和开发的领域值得思考。

10.3.1　DOM和模型操作

一个项目可以同时使用 jQuery 和 Angular。如果没有用到 jQuery，Angular 会使用内置的 jQuery 子集。在这个层面上，它们是兼容的，并且都支持 DOM 操作。

然而，两种框架都可以考虑时，应该优先考虑使用 Angular。这是因为每种框架在维护应用状态时都有本质上的区别。jQuery 应用通过直接操作 DOM 和控制视图的方式控制应用状态。Angular 应用却是将模型作为“真正的源”，而不是 DOM。框架会直接操作模型，而不是操作 DOM。模型的改变会直接导致 DOM 的改变。如果在 Angular 应用中依赖 jQuery 的 DOM 操作会导致一些难以发现的小问题，因为 Angular 的模型无法同步并且不知道通过 jQuery 做了哪些更改。

由于方法上的根本性差异，我们很难将 Angular 整合到一个已存在的基于 jQuery 的应用中。Angular 的 DOM 操作需要通过 Angular 指令来完成。将 jQuery 应用的功能转化为指令也很具挑战性，如果从最开始就使用 Angular 编写应用，一切都会简单得多。

10.3.2 Angular的不可见性

大家对 Angular 是否符合 JavaScript 在 HTML 中的不可见原则有着不同意见，但是在 MV* 框架中需要一个共同点将表现和行为进行连接。我们可以通过如下示例考虑这个问题。

将 JavaScript 代码放在 HTML 中会被认为是不合适的行为：

```
<button onclick='someFunction()'>Click Me</button>
```

在 jQuery 中，HTML 元素通过 CSS 选择器来定位，通过 ID 来选择一个元素是通用做法：

```
<button id='myButton'>Click Me</button>
```

通过 jQuery 元素选择器的事件处理程序对选择的元素进行事件绑定，以建立起 HTML 和 JavaScript 的连接：

```
$("#myButton").on('click', function(){someFunction();});
```

Angular 通过使用指令（HTML 的自定义属性）来实现这种连接：

```
<button ng-click="someFunction()">Click Me</button>
```

这并不违背 JavaScript 在 HTML 中不可见的目标。Angular 的属性具有明显优势：它们具有简单且易理解的含义。而 jQuery 选择器需要使用任意的 HTML 属性（ID 和类）。使用这些标准属性会引起歧义，我们不清楚这些值是否会影响表现或行为，甚至两者都影响。HTML5 引入了 data 属性，从而使这个问题得到适当缓解。

HTML data 属性

HTML 的 data 属性是指以“data-”开头且不会影响布局和展现的属性。这类属性专门用于存储数据。多数属性是通过属性名中“data-”之后的字符来区分的。

```
<li class="pet" data-name="Katniss" data-type="Civet" >
Hi Kat
</li>
```

10.4 项目

学习 JavaScript 框架有两种简单的方式。你可以从一个最小的可用项目开始，然后逐步对它进行扩展；你也可以从一个功能全面的引导项目开始，然后去填充组成项目的大量空文件。这两种方式在某些情形下都是有效且适用的。

最小可用示例更易于排除错误和交流问题（特别是在类似 JSFiddle 网站 http://jsfiddle.net/ 上的示例），而引导项目则会为全规模项目的后续开发提供稳定的基础。下面就是一个从最小可用项目开始，逐步增加功能的示例。

应用基于 JRuby 1.7.4、Sinatra 和 JSON gem 运行。在每一页中都迭代增加额外功能。

想要运行这个项目，先下载源码（https://github.com/java-javascript/client-server-web-apps/tree/master/Chapter-10-JRuby-and-Angular/rvm-jruby-sinatra-REST），安装 RVM 和任意版本的 JRuby，然后安装依赖的 gem，再运行服务器：

```
$ rvm list
rvm rubies

   jruby-1.7.4 [ x86_64 ]
=* ruby-1.9.3-p194 [ x86_64 ]
   ruby-2.0.0-p247 [ x86_64 ]

# => - current
# =* - current && default
#  * - default

$ ls
README.md    public     webapp.rb

$ ruby webapp.rb
[2013-08-13 21:20:01] INFO WEBrick 1.3.1
[2013-08-13 21:20:01] INFO ruby 1.9.3 (2013-05-16) [java]
== Sinatra/1.4.3 has taken the stage on 4567 for development...
[2013-08-13 21:20:01] INFO WEBrick::HTTPServer#start: pid=71632 port=4567
```

Web 应用的根目录是基于 HTML 文件的一系列公共目录的链接列表。虽然有点卖弄的嫌疑，但这其实是一个展现 Ruby 简洁性和表现力的有趣示例。下面代码中 `webapp.rb` 定义的 API 是关于其用途的描述。

```
get '/' do
  Dir.entries('public').entries.map{|f|
    "<a href='#{f}'>#{f}</a><br/>" if f=~/.*\.html/
  }.join
end
```

通常情况下，HTML 文件生成于服务器端，并且由单独的模板文件渲染而成；或者返回 JSON、XML 或其他格式的数据。Sinatra 本质上是一个 HTML DSL，对于返回值的类型没有特殊限制。图 10-3 显示了 Web 应用的目录，其中链接指向 Angular 示例。

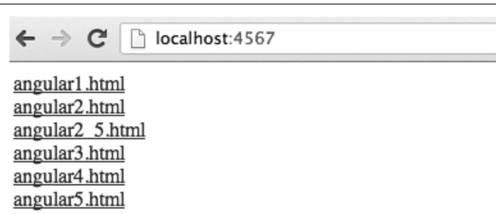


图 10-3：Web 应用的链接

第一个示例（如图 10-4 所示）简单地显示了 2 个文本框。如果改变其中一个文本框中的文本，相应的文本也会显示在另一个文本框中。

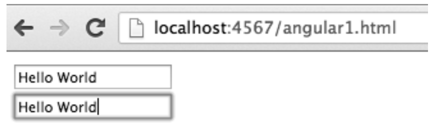


图 10-4：Web 应用的文本框

Angular JavaScript 文件（`script` 标签之后的内容）的压缩版本可以在 Google 的公共库（https://developers.google.com/speed/libraries/）下载。JavaScript 文件加载完成后，会遍历 DOM 以寻找 Angular 指令（HTML 自定义属性）。框架会识别出 `ng-app` 指令，并将它作为整个应用的外边界。`ng-model` 指令用来识别模型，比如改变一个文本框的值，另一个由于内置的双向数据绑定也会改变。

```
<!DOCTYPE html>
<html ng-app>
<head>
  <meta http-equiv="Content-type" content="text/html; charset=utf-8">
  <title>angular1</title>
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular.min.js">
</script>
</head>
<body>
  <input type="text" ng-model="myModel" value="{{myModel}}" />
  <br />
  <input type="text" ng-model="myModel" value="{{myModel}}" />
</body>
</html>
```

下一个示例将演示如何调用外部服务（在本例中是谷歌财经，即 Google Finance）和展示返回的 JSON 对象，如图 10-5 所示。Google Finance API 已经被废弃（http://googlecode.blogspot.com/2011/05/spring-cleaning-for-some-of-our-apis.html），但是具体的关闭时间还不确定。这个初始示例不太漂亮，功能也不太完整，但是只要少量的 Angular 功能就可以完成全部工作。

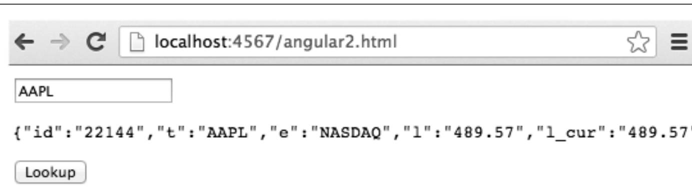


图 10-5：苹果电脑中的 Google 财经数据

综合实例中突兀的 JavaScript 代码

在这些例子中，JavaScript 代码包含在行内，在一处就能看到全部逻辑。根据不可见原则的规定，这样的代码应该是要分离的。此外，理想情况下 Angular 控制器也应该是轻量的，复杂的逻辑应该提取到 Angular 服务中。书中提到的这些例子都是用于帮助理解的单个文件，这种突兀是由于解释和说明需要造成的，在实际项目中则会更加优雅。

在前一个例子中，ng-app 指令单独放置。在这个例子中，angular 模型被特意命名为 app（在 HTML 元素上 ng-app 属性的值）。Angular 源文件被分离成几个不同的 JavaScript 文件，用于按需加载。Angular 库资源在 Angular 基础文件之后立即被加载，同时包括 ngResource 模块。这个模块允许 REST 式服务而不是调用低级的 $http 服务。我们创建的 app 模型是基于 ngResource 模块的。

在本例和上例中，ng-model 被映射到一个 JavaScript 变量。这里的 ng-controller 指令对应一个被称为 AppCtrl 的 JavaScript 函数。虽然所有的 Angular 控制器都是 JavaScript 函数，但反之就是错的。lookup() 函数被包含在 controller 函数的执行上下文中（或者说 $scope）。正如 ng-click 指令所定义的那样，按钮无论何时被点击都会调用对应的函数。lookup 函数会调用定义在 ngResource 模块中的 $resource 服务，参数会被传递给 Google Finance API 对应的服务。HTTP GET 请求使用 JSONP（因为所请求的服务是跨域的），得到一个返回的数组。返回的数组值被写入 $scope.result，其中第一个记录会以原始形式展现在前一个元素中。双花括号中的 Angular 表达式会被 $parse 服务处理并绑定。

```
<!DOCTYPE html>

<html ng-app="app">
<head>
  <meta http-equiv="Content-type" content="text/html; charset=utf-8">
  <title>angular2</title>
<script src=
"http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular.min.js">
</script>
<script src=
"http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular-resource.min.js">
</script>
<script>
angular.module('app', ['ngResource']);

function AppCtrl($scope, $resource) {

    $scope.lookup = function(){

      $scope.result = $resource(
        'https://finance.google.com/finance/info',
        {
                    client:'ig',
```

```
                        callback:'JSON_CALLBACK'
                },
                    {
                        get: {
                                method:'JSONP',
                                params:{q: $scope.stockSymbol},
                        isArray: true
                    }
                }
                                ).get();
                  }
        }
        </script>
        </head>
        <body>
                <div ng-controller="AppCtrl">
                        <input type="text" ng-model="stockSymbol" />
                        <pre>{{result[0]}}</pre>
                        <button ng-click='lookup()'>Lookup</button>
                </div>
        </body>
        </html>
```

可以将原始数据格式化之后进行展示。例如，可以使用以下方式展示实时价格：

```
{{result[0].l_cur}}
```

受篇幅限制，此处就不贴出所有代码了，完整的代码可以参考 GitHub（https://github.com/java-javascript/client-server-web-apps/tree/master/Chapter-10-JRuby-and-Angular/rvm-jruby-sinatra-REST）。

如图 10-6 所示的 angular3.html 增加了一个功能，从而可以将股票添加到以表格形式显示的“投资组合”中。我们可以通过一个输入文本框来过滤数据，还可以通过点击列头来对其进行排序。表格中的记录也可以被删除。遗憾的是，没有地方可以保存数据，所以页面刷新之后所有之前添加的记录都会丢失。

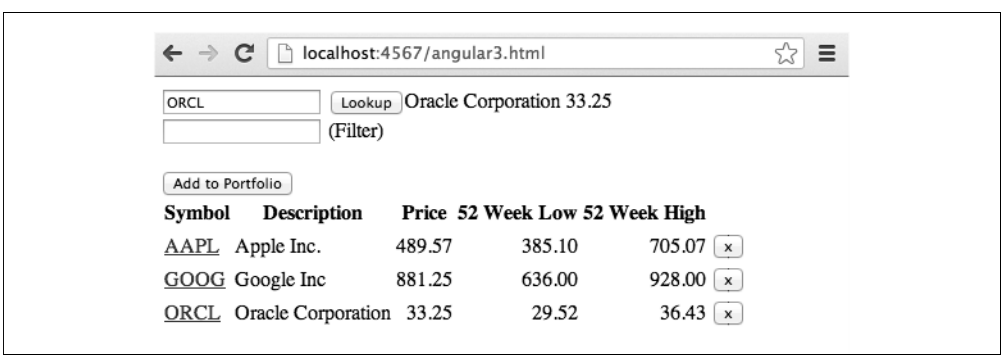


图 10-6：投资组合列表

示例 angular4.html 通过引入集成服务器端弥补了之前的缺陷。数据被存储到内存中，除非服务器重启，否则数据会一直存在。`webapp.rb` 是一个使用 Sinatra 微型框架构建的 Ruby 应用，前两行代码用来引入相关的 Ruby 和 Java 资源。严格来说，Java 可以完全省略，应用可以运行在基于 C 的 Ruby 组件之上。这里引入 Java 仅仅是用来演示 Java 是怎么被引用的。如图 10-7 所示，/version 的 GET 请求中包含了一个 `System.currentTimeMillis()` 的请求。

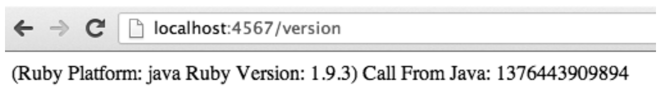


图 10-7：页面显示 Java 和 Ruby 都可用且能运行

一个类变量（`$stocks`）被定义成数组形式，用来记录股票的投资组合。HTTP PUT、DELETE 和 GET 请求用来实现对于单独股票记录的 CRUD 操作，而 /stocks GET 链接则会返回股票列表信息。

```
%w{rubygems sinatra java json}.each{|r|require r}
java_import 'java.lang.System'

$stocks = []

get '/' do
  Dir.entries('public').entries.map{|f|
    "<a href='#{f}'>#{f}</a><br/>" if f=~/.*\.html/
  }.join
end

get '/version' do
  "(Ruby Platform: #{RUBY_PLATFORM} "+
  "Ruby Version: #{RUBY_VERSION}) " +
  "Call From Java: #{System.currentTimeMillis()}"
end

get '/stocks' do
  $stocks.to_json
end

get '/stock/:t' do
  stock = $stocks.find{|e|e['t']==params['t']}
  if stock.nil?
    status 404
  else
    status 200
    body(stock.to_json)
  end
end
```

```
delete '/stock/:t' do

  stock = $stocks.find{|e|e['t']==params['t']}
  if stock.nil?
    status 404
  else
    $stocks.delete(stock)
    status 202
    body(stock.to_json)
  end
end

put '/stock/:t' do
  o = JSON.parse(request.body.read)['data']
  puts "---\n #{o['name']} \n---\n"
  $stocks << o
  status 200
end
```

示例 angular5.html 是我们第一次添加样式。如图 10-8 所示，这个示例引入了 Twitter Bootstrap CSS，其中添加了一些样式以及符合 Bootstrap 规范的 HTML 元素。



图 10-8：添加样式后的投资组合

重构这个项目，将 HTML 中的 CSS 和 JavaScript 提取到外部文件中会很有意义，整个项目可能会被改造成面向目标用户和开发者工具的引导项目。

10.5 小结

相比其他流行库，Angular 和 Sinatra 只需要更少的代码就能构建出高度动态的应用。它们值得研究不仅仅是因为它们自身的价值，更在于它们对其他技术产生的持续影响力。Sinatra 启发了很多其他 HTTP DSL。Angular 提前使用了一些类似 HTML5 Web 组件的技术，比如模板声明和双向绑定。它们既是可用于当前项目的成熟技术，同时也预示着 Web 开发的未来趋势。

第 11 章

打包和部署

“那天我在码头等人时，脑袋里突然跳出一个想法：不挪动车里的货物，而是连拖车一起吊放到船上，这样岂不是简单很多？”

——马尔科姆 • 麦克林

马尔科姆 • 麦克林在 1937 年的一个想法，让他后来成为了传奇的“集装箱化之父”，当时他正在霍博肯码头等待将自己的一包包棉花运往海外。1956 年，麦克林发明了金属运输集装箱，极大地简化了货物装卸，为航运业带来了一场革命。打包方式的创新提高了装运的效率。在科技领域，部署和发布应用犹如装运。Java 提供了标准的打包方式，这是发布代码、部署应用的基础。

11.1　打包Java和JEE应用

Java 的打包格式是 JEE 应用的基础构建模块，而且还被用在近来出现的、没有严格遵守 JEE 规范的部署流程里。

最初，Java 开发者在部署包之外编写代码。一个 Java 类对应操作系统中的一个文件。让人感到迷惑的是，Java 的包和打包方式无关，而是一个反映了从应用的根目录到类所在目录的命名空间。在程序员的开发环境之外，很少能看到类和包。应用或模块在部署到生产环境或最终用户之前，需要压缩并打包成一个单元。

Java *归档*（JAR）文件用来将 Java 类和资源文件打包成单个归档文件。JAR 文件使用 ZIP 格式压缩，并包含一个路径名为 META-INF/MANIFEST.MF 的 Manifest 资源配置文件。可

以使用 JDK 提供的 jar 实用工具（http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/jar.html）来生成 JAR 文件。简单来说，JAR 文件就是一个 .zip 文件，只不过在 META-INF 目录下包含了一些额外的描述信息。

JAR 并不专属于 JEE，它是标准 JDK 的一部分，其规范（http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html#JAR Manifest）可以在 JDK 文档中找到。JEE 描述了 JAR 文件的几种使用方式。JEE 应用客户端模块和 EJB 模块被打包成 JAR 文件。JEE 的其他打包格式也基于 JAR。根据功能和内容的不同，JEE 其他归档文件其实也是 JAR 文件，只是文件名后缀不同而已。我们可以手工创建 JEE 归档文件，但这并不常见，通常是使用诸如 Ant、Maven 或者 Gradle 这样的构建工具。

Web 模块是 JEE 中最小的部署单元，该模块包含 Web 组件和资源文件，主要是静态内容。Web 模块可以被打包成一个 Web 归档文件，或称为 WAR。WAR 文件有一个 /WEB-INF 目录，该目录下有一个名为 web.xml 的文件，它定义了 Web 应用的结构和对归档文件中其他资源的引用。WAR 格式极其灵活，可以被部署到只支持很小一部分 JEE 规范的 Web 容器中。

常用的 Web 容器有 Tomcat（http://tomcat.apache.org/）和 Jetty（http://www.eclipse.org/jetty/）。Web 容器的开发早于 JEE 规范里对各种技术的定义，因此每个容器都有各自不同的特性。

企业归档文件，或称为 EAR 文件，包含了 WAR、JAR 和一个 application.xml 文件，该文件引用了包含的模块，定义了安全角色。就像它的名字一样，EAR 为企业级应用而生，必须部署在应用服务器中。和 Web 容器相比，应用服务器支持更多的 JEE 规范。EAR 文件不能在 Web 容器里运行，它们需要 EJB 支持和一些其他服务。应用服务器有 JBoss（http://jbossas.jboss.org/）、IBM 的 WebSphere（http://www-01.ibm.com/software/websphere/）和 Oracle 的 WebLogic Server（http://www.oracle.com/us/products/middleware/cloud-app-foundation/weblogic/overview/index.html）。

为了描述的完整性，还得提一下另一种不太常见的 JEE 归档文件：资源适配器模块（RAR）。RAR 用来连接企业信息系统（Enterprise Information System，EIS）。企业信息系统通常指遗留系统，比如 ERP、主机、队列或其他服务。RAR 和 JDBC 驱动类似，为后台系统提供统一接口，但和 JDBC 不同的是，它不局限于连接关系型数据库。表 11-1 列举了相关的文件后缀。

表11-1：Java打包格式

后缀	名称	描述
.jar	Java 归档文件	标准 Java 打包格式
.war	Web 归档文件	映射至唯一的 Web 根路径
.ear	企业级应用归档文件	包含若干个 WAR 和 JAR
.rar	资源适配器模块	和企业信息系统通信

Web 应用开发者对 WAR 尤其感兴趣，其他语言的框架也纷纷采用这种格式，由此可见一斑。Play 框架创建的 WAR 包含 Scala 资源，Ruby 有一个名为 Warbler（http://caldersphere.rubyforge.org/warbler/）的 gem，专门用于将 Ruby 应用打包成 JAR 或 WAR 文件。

现有的 Java 和 JEE 打包方式已经能够很好地用于开发客户端 – 服务器端的 Web 应用。开发稍具规模的项目时，如何将服务器端和客户端的代码分离到各自的归档文件中是我们需要考虑的。一个应用可以被部署为一个 EAR，其中包含提供 API 代码的服务器端的 WAR 包，以及包含 HTML、CSS 和 JavaScript 的独立的客户端 WAR 包。这两个 WAR 包也可以脱离 EAR，分别部署到两个 Web 容器中。和我们目前讲到的其他领域不同，打包方式一直没有什么重大创新。

WAR 包示例

可以使用 Maven 构建本章所附代码（https://github.com/java-javascript/client-server-web-apps/tree/master/Chapter-11-Packaging-and-Deployment/jersey-jquery-REST）从而生成随后插图中使用的 WAR 包。该 WAR 包包含了客户端和服务器端的代码，只做最简单的展示之用。

归档文件作为一个打包了相关 Java 资源的独立包，在分离依赖方面做得并不好。归档文件（假设是自包含的）需要部署到一个运行环境中执行。尽管打包方式一直以来变化不大，但部署方式却改变了很多。

11.2 JEE应用的部署

过去，部署方式十分有限，各种方式的区别主要在于自动化程度。部署应用时，总是假定已经有程序员或系统管理员安装并配置好了应用服务器。直到今天，JEE 规范本身仍是朝着这个方向，从其对参与部署应用的角色描述上就可见一斑。JEE Web 应用的部署方式如图 11-1 所示。

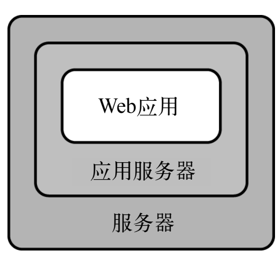


图 11-1：JEE Web 应用的部署

JEE 中的角色描述了参与开发流程的人的分工，包括部署者和系统管理员。部署者负责在

运行环境中配置应用、验证模块是否遵循 JEE 规范，以及在一台或多台服务器上安装应用模块。从描述上可知，人们希望已经有人或组织在 JEE 应用的目标环境中安装好了应用服务器。（和大规模部署中持续交付的通用做法不同，它强调了部署时的人工干预；关于“持续交付”，请参见 http://en.wikipedia.org/wiki/Continuous_delivery）就像前面说的，这并不是部署 Java Web 应用的唯一方式，但却是唯一遵循规范的方式。

JEE 和云部署

和其他巨大的投入一样，JEE 也力求和以前版本保持连续。同时，它也在试图覆盖一系列部署环境。有些情况下，这是它的优势，有时却成为了缺点。最近一次 JEE 版本的更新又增加了云部署方式，虽然部署者的职责看似被削减到最少，但在实践中，部署者也常常是系统管理员。

> 以云部署为例，部署者负责配置应用，让其在云环境中运行。部署者在云环境中安装应用、配置其外部依赖，可能还要准备需要的资源。
>
> 在云环境下，系统管理员负责安装、配置、管理和维护云环境，包括供应用在环境中运行使用的资源。
>
> ——JEE7（http://download.oracle.com/otndocs/jcp/java_ee-7-fr-eval-spec/index.html）

若要遵循 JEE 规范，事先安装好应用服务器，只有有限的几种部署方式，但这并不代表全部的部署实践。可以在待部署的服务器上构建项目，这要求所有相关服务器上都安装有构建工具，但会导致系统性能下降或中断服务器上的服务。如果在生产服务器上安装额外的软件或者部署未经测试的代码，则会带来潜在的安全漏洞。这些问题都意味着最好不要在生产服务器上构建项目，而是将打包好的应用分发给生产服务器来进行部署。

随着在大量服务器上部署的现代部署方式的出现，在部署服务器上构建项目的情况越来越少。极端情况下，这可能有用，但总体上来说，最好在一个非生产机器上构建，然后将结果分发给目标服务器进行安装。即使在项目早期不必如此，但它提供了很大的灵活性，如果应用的访问量逐渐增长，就有必要增加新的服务器了。近些年，很多分布式远程执行 Shell 命令的项目被开发出来，让这种部署方式变得更容易管理。

11.2.1 图形界面管理

用户可以通过与图形界面交互来管理应用服务器，虽然并不是所有的 Web 容器都提供了这种管理方式。在很久以前，很多图形界面还是那种老式的、本地原生应用，通过网络远程连接至应用服务器。现在，GUI 都是单独的 Web 应用。应用安装好后，需要进行额外配置。需要防止管理门户被公开，这可能会造成一个安全漏洞；需要调整管理服务器的根目

录，避免和其他部署其中的 Web 应用冲突。这些配置方面的考虑让很多管理员在初始安装过程中直接禁用管理 Web 应用。

红帽公司发布的 JBoss 企业应用平台 6（JBoss Enterprise Application Platform 6，JBoss EAP 6）就自带一个基于 Web 的图形管理界面，如图 11-2 所示。下载（http://jbossas.jboss.org/downloads）并添加一个管理账户后，启动服务器，默认显示管理界面。点几个按钮，管理员就能部署和激活一个 WAR。这样一个 Web 应用就部署成功了，根目录也指定好了。

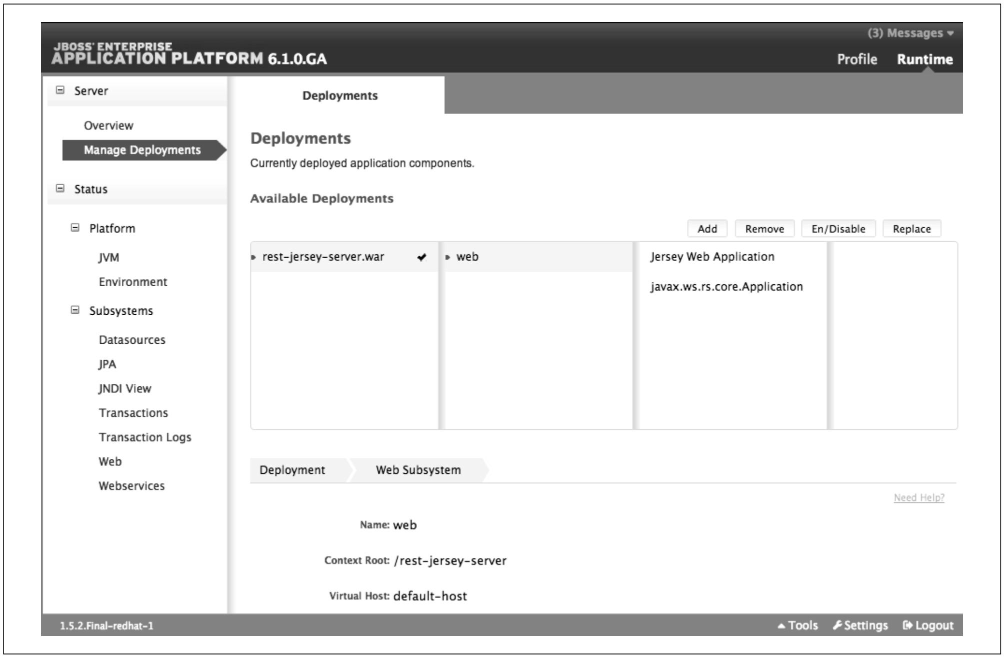


图 11-2：JBoss Web 管理界面

JBoss 是一个功能齐备的应用服务器，打包了很多有用的模块，这可以最大限度地减少你需要打包到一个 WAR 包里的代码。但当你的应用里有和 JBoss 相似的模块时，也会产生一些不易发现的错误。本章提供的 WAR 文件包含了 Jersey 和它的依赖。在部署过程中，包含 Jersey 会出错。RestEasy（包含在 JBoss 中）会报错，默认情况下，它会扫描并认定 Jersey 是一个实现了 JAXRS 规范的冲突。解决方案是为 web.xml 添加上下文变量以关闭扫描：

```
<context-param>
    <param-name>resteasy.scan</param-name>
    <param-value>false</param-value>
</context-param>
<context-param>
    <param-name>resteasy.scan.providers</param-name>
    <param-value>false</param-value>
```

```
    </context-param>
    <context-param>
        <param-name>resteasy.scan.resources</param-name>
        <param-value>false</param-value>
    </context-param>
```

解决方案虽然简单，但却指出了问题所在：尽管 JEE 规范明确定义了打包机制，但依然不确定哪些服务应该被包含进一个特定的部署环境。这些特质让部署 JEE 应用时“一次编写，随处部署”这句话只在极为严格的约束下才成立。

11.2.2　命令行管理

初始安装应用时，使用 GUI 管理很方便。所有可用的管理选项都一览无余地显示在用户界面上，很容易理解。这让图形管理界面成了学习应用服务器和检查可用功能的理想之选。和所有 GUI 一样，与命令行相比，它们没有编写脚本并自动化执行的功能。对于小规模部署，命令行只是增加了一点便利，但对于复杂的部署（或者拥有大量服务器的简单部署）场景来说，命令行就是必不可少的了。

JBoss 有一个命令行接口，可以通过在命令行提示符下输入指令或执行脚本来进行管理。几乎所有在 GUI 上可见的操作都可以通过命令行完成。将命令行接口连接到一个正在运行的应用服务器上就可以初始化一个命令行会话。为此，需要调用 `cli`，使用 `-c` 选项（立即建立一个到应用服务器的连接）：

```
$bash bin/jboss-cli.sh -c
```

登录成功后，可键入 `help` 命令获取可用命令列表：

```
[standalone@localhost:9999 /] help
```

只需会一些基本命令就可以执行大多数的常用操作。使用 `ls` 命令列出给定节点路径下的所有内容：

```
[standalone@localhost:9999 /] ls
```

这样，在 JBoss 环境下，就可以像在操作系统的文件系统里那样，使用 `cd` 命令在目录间跳转，使用 `ls` 命令列出给定节点路径下的的内容。可用如下方式查看部署好的 WAR 包：

```
[standalone@localhost:9999 /] ls deployment
rest-jersey-server.war
```

该命令行的语法并不完全和操作系统中的一致。比如，可以使用 `ls` 命令和等号来指定部署物以获取 WAR 包的更多信息：

```
[standalone@localhost:9999 /] ls /deployment=rest-jersey-server.war
```

JBoss 的 CLI 命令被设计为可在外部脚本中执行。下面的命令展示了如何在操作系统的命令行窗口中打印当前部署列表：

```
$bash bin/jboss-cli.sh -c --commands='ls deployment'
```

CLI 让编写复杂脚本和应用服务器、操作系统及其他应用交互成为可能。上述例子只展示了查询操作，但 CLI 的功能不止这些。它还可以修改应用服务器状态，比如部署或取消部署一个 WAR 包：

```
[standalone@localhost:9999 /] undeploy rest-jersey-server.war
[standalone@localhost:9999 /] deploy /tmp/rest-jersey-server.war
```

对于很多管理任务，管理员都可以将 GUI 人工交互的方式替换为 CLI 脚本，但 CLI 有自己的语法和组织方式，需要花时间学习才能有效使用。如果只关心如何部署一个应用，那么既不必使用 GUI，也不必使用 CLI，直接将文件复制到指定部署目录下即可。JBoss 的*部署扫描仪*会扫描该目录，自动部署应用。复制操作可以手动，也可以使用脚本，因此即使不学习 CLI，也能自动部署。需要经常进行部署的开发者可选择这种复制的方式，依赖部署扫描仪或者集成进他们标准工具链里的钩子程序。人们为 Maven 和其他构建工具以及 IDE 编写了插件（http://docs.jboss.org/jbossas/7/plugins/maven/7.4.Final/examples/deployment-example.html），将应用部署集成进构建流程里。

11.3　非JEE应用的部署

长久以来，Java Web 开发等同于 JEE（Java 企业版）。Java 开发者吃惊[1]地发现 JEE 模型并不是 Web 应用部署的唯一方式，而且对于特定系统，甚至可能还不是最好的部署方式。Java 技术不局限于开发 Web 应用，JEE 也只是在 Java 平台上开发 Web 应用的可能方式之一。图 11-3 展示了 Java Web 应用的开发模型。

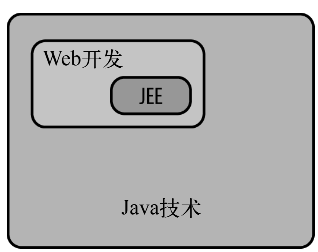


图 11-3：使用 Java 开发 Web 应用

注 1：原文是“It can be jarring (pun intended) for Java developers to realize”，作者使用了形容词 jarring，一语双关，同时还有 Java 中打包的命令 jar。

部署一个非 JEE Web 应用时，不需要事前安装应用服务器。从 Web 应用的角度看，应用服务器提供的上下文环境在外部、内部或与之并行。这并不是应用服务器特有的选项，其他提供独立服务的应用（比如数据库）也是如此。

11.3.1 服务器在应用之外

应用服务器是一种复杂且成熟的软件，已经面世很多年了。过去，受磁盘和处理器限制，以及配置的复杂性，人们不得不花费大量时间和精力安装它。安装好后，才能安装和配置 Web 应用连接已有服务。如图 11-4 所示，Web 应用是跑在服务器“上”的。应用服务器是 Web 应用的部署目标，其本身处在 Web 应用“之外”。

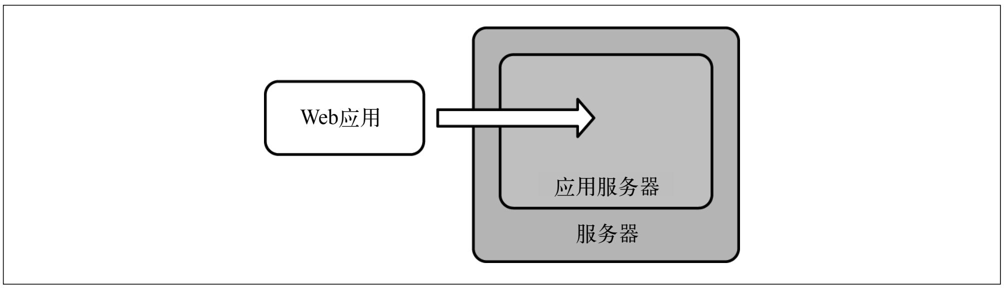


图 11-4：应用服务器在 Web 应用之外

这事实上就是本章前面讲述的 JEE 的部署方式。当你需要将应用部署至已运行应用服务器的内部系统中，或是希望将多个 Web 应用和 JEE 包打包进一个 Web 应用时，这种方式很好用。很多时候，应用不需要什么管理，可能就需要放在一个应用服务器或者 servlet 容器里来运行，但是不需要任何管理。这就让开发者在部署自己的 Web 应用时，才想起部署应用服务器。

使用轻量级容器成为趋势

应用的虚拟化和轻量级部署是大势所趋，这就需要抛弃庞大的、整体式且需人工安装配置的应用服务器。基于云的部署方式的出现让服务器的概念变得模糊了。这种方式要求部署速度快、尽可能简单，并且高度自动化。

11.3.2 服务器和应用并行

不事先安装应用服务器的情况下，也有几种方式可以部署 Web 应用。一种方式是将 Web 应用和应用服务器捆绑在一起。应用服务器是独立的，部署时和 Web 应用并行安装在一起，如图 11-5 所示。

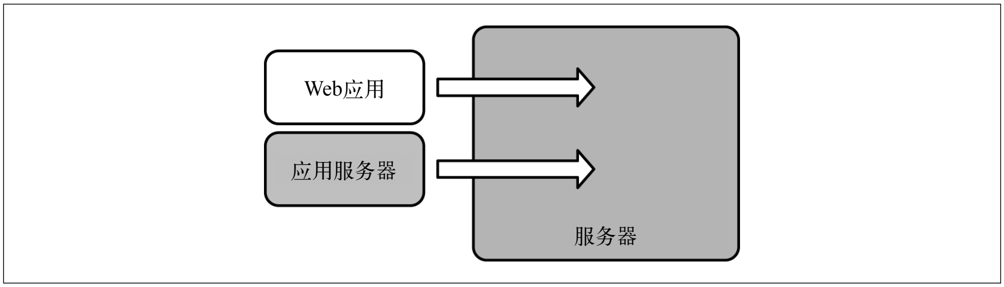


图 11-5：服务器和应用并行

这种部署方式在 Rails 里很流行，它本身包含一个服务器（WEBrick，http://www.ruby-doc.org/stdlib-2.0.0/libdoc/webrick/rdoc/）和框架。Play 和 Roo 从 Rails 那里得到启示，也使用这种部署方式，不过要依赖 Java Web 容器。Maven 的 Jetty 插件（如图 11-6 所示）是另一个示例，Web 应用部署后，可立即在 Servlet 容器中运行，前提是不需要外部维护和管理。使用本章提供的项目，读者可自行构建 Web 模块，并使用下述命令在 Jetty 上运行生成的 WAR 包：

```
mvn clean install jetty:run
```



图 11-6：运行在 Maven Jetty 插件中的 Web 应用

通过端口 9090 就可以访问该应用了。

使用 Jetty 时，Maven 并不需要运行 Web 应用。可以用 Jetty Runner（http://wiki.eclipse.org/Jetty/Howto/Using_Jetty_Runner）将 Jetty 打包进一个 JAR，在命令行下使用。在 build 目录下，将路径和 WAR 作为参数传给 Jetty Runner：

```
curl -O http://repo2.maven.org/maven2/org/mortbay/jetty/jetty-runn
er/8.1.9.v20130131/jetty-runner-8.1.9.v20130131.jar

java -jar jetty-runner-8.1.9.v20130131.jar \
--path /jersey-server \
target/rest-jersey-server.war
```

Tomcat Runner（https://devcenter.heroku.com/articles/java-webapp-runner）是另一个使用 Tomcat 的类似项目（https://github.com/jsimone/webapp-runner）。它的参数和使用 Jetty Runner 时一样：

```
java -jar webapp-runner-7.0.40.0.jar \
--path /jersey-server \
target/rest-jersey-server.war
```

11.3.3　服务器在应用里面

除了使用一个外部服务器，还可以在应用代码中包含一个 Java 服务器库，如图 11-7 所示。服务器是在 Web 应用里面运行的。第 6 章有几个案例，展示了可这样使用的几个库的框架。

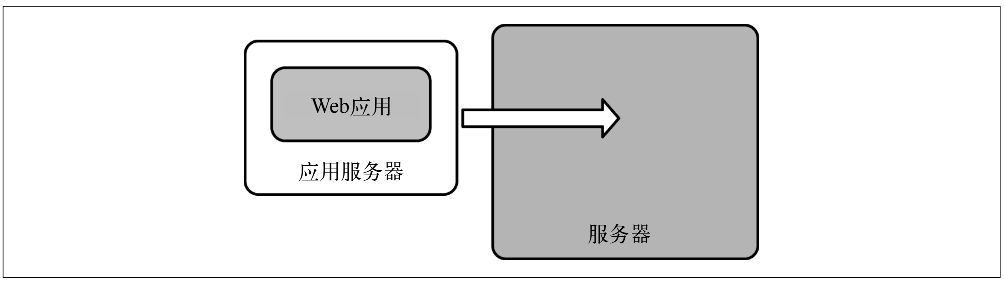
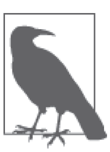


图 11-7：服务器在 Web 应用里面

使用这种将服务器包在应用里的方式，发布时仅需创建和部署一个可执行的 JAR 包，就可以放在服务器上运行。JEE 标准并没有考虑到 JAR 或应用服务器可以这样使用，而是像前面描述过的，假设应用会部署到一个已经配置好的环境中，提供了诸如 HTTP 处理、数据库连接这样的服务。



合并可执行的 JAR 包

尽管在 Java 项目里引用多个 JAR 包的情况很常见，但是一个项目也能以一个单独的可执行单元进行打包和部署。将 JAR 包打包进另一个 JAR 包的工具有 one-jar（http://one-jar.sourceforge.net/），还有构建工具的插件，比如 Maven Shade plug-in（http://maven.apache.org/plugins/maven-shade-plugin/）。

11.4　不同部署方式带来的影响

部署方式的选择会对应用的安全性、扩展性及整体支持带来显著影响。应对快速变化时，不同的部署方式在灵活性上有所差异。

或许可以在单个服务器上简单部署展开的 EAR 或 WAR 包，为系统打热补丁，这样即使更新代码或配置文件，也不需要再重新完整地部署一遍。这种方式会引发大量安全问题，但

更致命的是，在复杂的部署场景下，这种改动是行不通的。

应用服务器在外部管理时，很容易修改数字凭证和连接字符串。而当应用和应用服务器并行部署或将应用服务器打包进应用时，则不会这样。

应用服务器不依赖外部管理的部署方式为水平扩展提供了更大的灵活性。通过构建智能的负载均衡器，不用改变现有服务器，而是建立新的并且按期望配置好的服务器，让人们能以最短的宕机时间快速响应变化。

部署方式的不同也会影响开发流程和对应用基础的选型。最好在开发流程初期就确定好部署目标，以确保所需资源的就位和相关流程的制订。

11.4.1 负载均衡

负载均衡和部署方式紧密相连，它的实现方式决定了运行应用的网络和服务器拓扑结构。负载均衡的目的是在现有处理器能力下，尽可能高效地分发进来的请求。以 Web 应用为例，进来的 HTTP 请求经指定的负载均衡服务器（或服务器集群）重定向到多台 Web 服务器。

根据处理每个请求的工作量、服务器的处理能力和执行负载均衡操作所选择的硬件或软件的不同，分发请求的方式也不同。可以采用轮询方式在服务器之间平均分配任务；也可以根据权重将更多任务分配给处理能力更强的服务器。还有更复杂的调度方案，有的会记录每台服务器处理的请求，有的则让服务器在空闲时自动拉取任务执行，这些方式均能更好地利用现有的处理器能力。有一些负载均衡器能检测出失效节点，不会将请求分发到这些节点上。负载均衡的工作机制如图 11-8 所示。

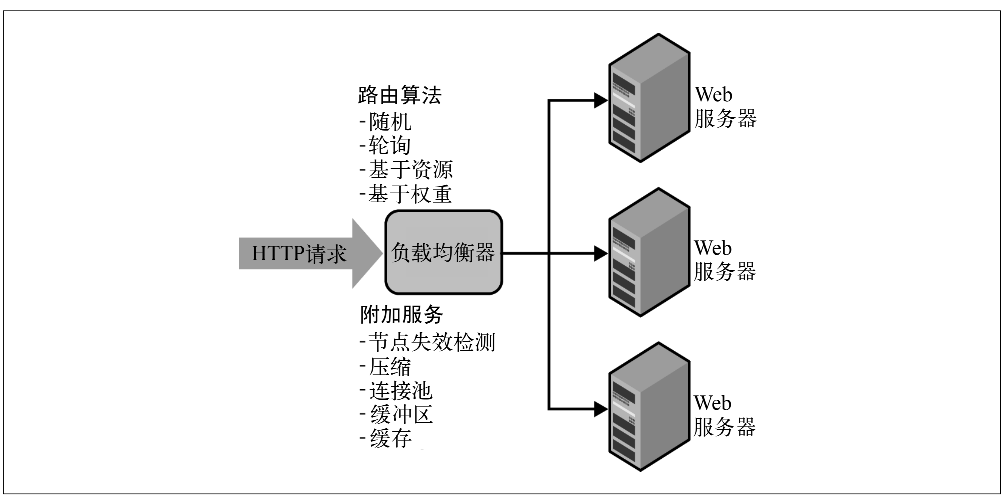


图 11-8：负载均衡的工作机制

不是所有物理负载均衡器的功能都能被软件负载均衡器替换。这其中包括一些安全方面的考虑，比如分布式拒绝服务防范和 SSL 终止，还有一些有助提升性能的特性，比如压缩、连接池、缓冲区和缓存。这些功能并不一定是负载均衡所特有的，但有时高效地使用它们非常必要。

更多关于负载均衡的介绍

负载均衡的很多方面其实和客户端－服务器端的 Web 开发无关。JBoss 有一个内部的负载均衡（http://jbossclustering.jboss.org/），在集群环境下为 JNDI、RMI 和 EJBs 提供服务。它和其他许多 Web 容器都包含集群功能，让会话在多台应用服务器上都是可用的。根据是否将来自同一个客户端的多个请求定向到一台服务器，不同的负载均衡有不同的策略。基于 DNS 的负载均衡和轮询的方式类似，但是客户端缓存了服务器 IP，在第一次查找后会返回同一个服务器。粘滞会话将和同一个会话关联的 HTTP 请求发送至同一个服务器。负载均衡的范围很大，从对负载均衡器的内部管理的独立操作到 AWS Elastic Load Balancing（http://aws.amazon.com/cn/elasticloadbalancing/）这样的云部署方式。你可以翻阅有关它们的文档（http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/elastic-load-balancing.html）和过去对基础技术（http://oreil.ly/server-load-balancing）的讨论，从而获取更多信息以深入了解负载均衡。

开发者即使不直接参与负载均衡，了解足够的知识去设计和制订部署方案也是非常重要的。无状态的处理不再要求会话，这是需要着重调整的地方。安全配置也会受到影响。后续支持需要理解应用日志。如果不知道网络是如何建立的，就不能有效地利用日志中的请求和响应记录来诊断故障。

应用服务器集群

应用服务器或 Web 容器集群可以代替负载均衡。它能分散服务器端应用的负载，从而达到同样的目的。其实现细节没有标准化，也并非适用于所有服务器。集群在商用项目上尤其常见，但和其他企业级解决方案一样，很贵。负载均衡可能也会引入昂贵的硬件和软件，但不是非得如此。最基本的负载均衡原理已经被大家掌握，可以使用通用的、和厂商无关的网络配置（防止在不同应用服务器间厚此薄彼）来实现。

11.4.2 自动化应用部署

为了方便于控制可能部署场景的复杂性，人们开发出了很多工具。早在 20 世纪 90 年代，就有针对 cfengine 的专用脚本让工作站的管理实现自动化。它是一种非常轻量级的工具，这么多年来一直有人在维护。它支持多种操作系统（包括 Windows、Mac OS X 和其他一

些操作系统），这将它和那些为特殊 Web 应用定制的工具区分开来。

Capistrano 是一种在多台远程机器上并行执行 SSH 命令的实用工具。它是基于 Ruby，并且面向 Web 应用的部署，这和它来源于 Ruby on Rails 的生态系统是分不开的。Capistrano 的典型应用之一是从 SCM 系统中签出 Web 应用，将其部署至多台远程服务器上。Fabric 是 Python 版的 Capistrano。由于云部署环境下服务器的透明性，这些工具常和 Puppet、Chef、Ansible 和 Salt 诸如此类的配置管理工具放在一起考虑。

11.5 项目

前面已经展示过如何部署本章的项目。这个项目是一个简单的客户端 – 服务器端架构的 Web 应用，提供 CRUD 操作，用来添加、更新和删除图书。这个例子没有数据存储功能，用了一个数组保存添加的图书，当服务器重启时，数据就丢失了。

应用的架构反映在 pom.xml 和 web.xml 中。pom.xml 包含和 Jersey 相关的模块，将它作为依赖实现了 JAX-RS 风格的 API，输出 JSON。web.xml 区分开 API 的客户端和服务器端，并指明 index.html 为欢迎页面，是应用客户端的入口。

11.5.1 客户端

index.html 包含和应用相关的 HTML、CSS 和 JavaScript。通过 CDN 的方式提供 jQuery。JavaScript 代码内嵌在 script 标签里。jQuery 的 `getJSON` 和 `ajax` 方法用来调用 API。jQuery 的 `$( document ).ready()` 函数调用 `list` 函数，该函数以 JSON 格式获取图书列表，然后在该列表上迭代，将每个条目放到一个 `div` 里显示（该 `div` 通过 `delete` 类锚定），最后加到用 CSS 类 `listing` 查询到的段落对象后。DELETE 操作绑定到全局文档对象（而不是每一个 `div`）上，以确保所有属于 `delete` 类的超链接都能响应事件（如果直接和 `delete` 类绑定，那么只能和页面初始化时存在的元素绑定）。Bootstrap 也是以 CDN 的形式引入的。为保持简约，该应用并没有在很多地方使用它，把它包含进来作为起点只是为了增加一些样式。

这是一个最基本的实现，但已经足够说明这种基于 DOM 操作、没有 MV* 框架，也没有使用 CSS 框架的前端架构的局限性。

首先，使用了一个内嵌的样式将页面居中，这是非常不 Bootstrappy 的方式。Bootstrap 和其他响应式 Web 框架依赖基于网格的布局方式。修改 HTML，让其使用网格，从而更适合在不同的设备上显示。

另外，DOM 操作很难可视化。在加载一些记录后，HTML 页面的初始状态已经不能严格代表 DOM 的状态，这就表明一个视图化的模板系统可能是有用的。除了笨拙地将 JavaScript 代码嵌入页面，也没有按功能将代码划分成块。MV* 框架为应用提供了初始架构，可以更好地管理 JavaScript 代码。

11.5.2 服务器端

所有 Web API 都包含在一个 `BookService` 类中，在 web.xml 映射到 /api 这个 URL。类最上端的 `@Path` 注释表明该类被 /books 引用。GET、PUT 和 DELETE 方法用来读取、创建和删除图书资源。`@produces` 注释表明每个 JSON 调用都返回 `application/json` 的格式。在方法层面，`@Path` 注释引用的 `{book}` 映射为 `@PathParam`，传递相应方法的参数。

通常要使用库来创建 JSON，本例使用了 Mention Apache IO Utils 连接字符串创建 JSON。API 的调用可以在浏览器里验证和测试，也可以使用如 Curl 这样的工具。在 Maven 插件定义的上下文中，可以通过下述命令插入一条记录：

```
curl -X PUT http://127.0.0.1:9090/api/books/4?title=Client+Server+Web+Apps
```

11.6 小结

过去几年，遵循 JEE 流程的 Java 应用部署方式已经得到了很好的定义。虽然 JEE 风格的部署方式在很多情况下还是适用的，但 Java 社区还是采用了新的部署方式，尤其是对于客户端 - 服务器端风格的 Web 应用。下一章将讲述新出现的、应用于云环境中的虚拟化技术，它表明在很多情况下，将应用服务器打包到应用中或者和应用并行，比传统的将应用服务器放在应用之外的方式要好。

第 12 章

虚拟化

人们不懂得欣赏事物的本质，太空中的物体，人们忘了本质。

——具伯 · 尔利（Firefly[1]，Objects in Space）

一提起“计算机”这个词，人们脑海中会立刻蹦出一堆实体机形象，可能是一台显示器和一个键盘、一台笔记本电脑，或者是一排排服务器。不管是哪种情况，都是一种有形的、具体的实物。在软件开发中，这种对计算机的普遍印象就不是特别有用了，因为硬件的物理细节被一层抽象的软件给屏蔽了。虚拟化描述了一种隐藏实现细节的技术，它被广泛应用于硬件品台、操作系统、存储设备和网络资源中。虽然这和我们本书所讲的客户端 - 服务器端 Web 开发方式或者其他开发方式不直接相关，但却是一个有趣的话题，因为很多应用都需要大规模部署，而大规模部署需要建立在虚拟化解决方案上。虚拟化是个强大的概念，它影响应用开发、部署、扩展和灾难恢复的方方面面。

12.1　全虚拟化

没有虚拟化，服务器就定义和受限于物理约束。服务器管理员根据硬件情况和实际需求，构建和配置有一系列特定选项的服务器。开发团队中的成员参照目标服务器的配置，为自己的机器安装同样的软件，进行同样的配置。如遇硬件限制、性能瓶颈或容量问题，一般通过增加硬件来解决。有时也会用到一些自动化手段，但多数情况下，由于硬件的差别，日常工作主要靠手动完成。如果有大量的不同服务器需要维护，复杂的备份和恢复行动就

注 1：详见 http://en.wikipedia.org/wiki/Firefly_(TV_series)。具伯 · 尔利是剧中人物 Jubal Early。

成了保持系统正常工作和可靠性的当务之急。

通过用虚拟机替代物理服务器的虚拟化方式，可以在某种程度上缓解这种挑战。全虚拟化试图提供一套完整的、从基础硬件到运行在虚拟机上的独特且性能未改的操作系统的仿真。图 12-1 展示了虚拟化技术的发展，从传统的、持久的、需要手动配置的物理硬件到高度透明的虚拟机，这些虚拟机是自动创建的，有些甚至只在云中短暂存活几秒钟。

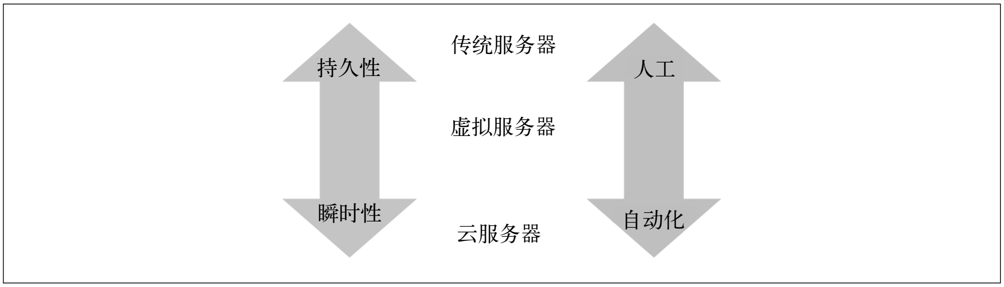


图 12-1：服务器类型

使用虚拟机可降低维护物理服务器带来的挑战。由于全虚拟化技术完全模拟了物理硬件，每个开发者都拥有独立的环境进行新软件测试，而不用花费金钱购买和维护额外的机器。一台物理服务器上可创建多个虚拟机，安装不同的操作系统以测试软件。所有需要安装的软件都可以在虚拟机里事先配置好。可以随时为机器当前状态创建快照的能力为灾难恢复提供了很多可能。与管理大量的、硬件千差万别的物理机相比，将服务器替换为虚拟机既减少了能源消耗，又降低了硬件成本。

然而虚拟机并不是一本万利的，它需要额外的处理。首先，必须购买虚拟化软件；其次，运行它们需要占用额外的处理器资源。对于那些需要访问底层硬件的项目来说，虚拟化并不适用，但对于大多数 Web 应用，虚拟化是个很好的选择。虚拟化和物理硬件的关系有点像高级语言和低级语言，比如 C 或者汇编语言。对性能要求高的任务需要直接对“金属裸机”编程，使用特殊的、高度定制的方案，而其他为数众多的任务则没有如此严格的要求。

另一个代价是需要考虑额外的复杂性。和关注物理服务器不同，虚拟机的行为不能割裂开来理解，而是需要和宿主机器结合起来考虑。性能问题可能出自虚拟机，也可能出自宿主机器。虚拟层可嵌套，这进一步增加了复杂度。虽然虚拟化解决了很多问题，但是需要从技术层面去了解它，学会如何高效使用。

虚拟机代替物理服务器的流行让服务器管理和扩展实践发生了巨大变革。所谓的云计算提供商以各式各样的方式提供了大量虚拟化的计算资源。有时，虚拟化的形式是提供具体的机器，这时就不需要管理物理服务器了。

JVM 中的“V”

Java 虚拟机（Java Virtual Machine，JVM）提供了部分的虚拟化，允许包含字节码的类文件在任何运行虚拟机的硬件上解释和执行。但它没有提供分隔代码的容器，因此无法在网络上创建可作为独立服务器进行管理的独特机器。任何形式的虚拟化都有效地屏蔽了底层实现细节，只是根据问题的不同，实现的层次也不同罢了。

12.2 虚拟机的实现

虚拟化技术要追溯到 20 世纪 60 年代 IBM 公司的一个研究项目：CP-40。1967 年 IBM 推出了它的后继产品 CP 67，这是一个为 IBM System/360-67 开发的虚拟机操作系统。此后几年，虚拟化解决方案不断涌现，大多数都是针对特定的操作系统。随着越来越多的开发者能以低廉的价格获得处理能力更强的硬件，虚拟化技术也变得流行起来。

在当今众多的虚拟机实现里，VMWare、VirtualBox 和 Amazon EC2 无疑是最受欢迎的。它们也是使用类似 Vagrant（http://docs.vagrantup.com/v2/why-vagrant/index.html）和 Packer（http://www.packer.io/）这样的工具来创建服务器时支持的目标平台。

12.2.1 VMWare

很多 Web 开发者碰到的第一个虚拟机都是在 1999 年发布的 VMWare（https://www.vmware.com/）工作站。现在的 VMWare 提供从云管理、备份到桌面产品相关的一系列虚拟化方案。另外，你可从网上下载一个开源版（https://my.vmware.com/web/vmware/info/slug/desktop_end_user_computing/vmware_workstation/10_0#open_source）的 VMWare。

12.2.2 VirtualBox

2007 年 1 月，开源版本的 VirtualBox（https://www.virtualbox.org/）发布，这是一个可以运行在 Windows、Linux 和 OS X 上的全虚拟化解决方案。不久后它就被 Sun 收购，随后 Oracle 又收购了 Sun，将其重新命名为 Oracle VM VirtualBox。它在常用功能上和 VMWare 类似，但在一些非技术功能上优势明显，比如友好的软件授权、付费功能、易用性和文档。

12.2.3 Amazon EC2

Amazon Machine Image（AMI）是定义服务器配置的模板，它运行在 Amazon Elastic Compute Cloud（Amazon EC2）之上。启动一个实例时，选择一个 AMI，随后它成为云中的一个虚拟服务器，因此，AMI 仅适用于针对 Amazon Web 服务的部署。

12.3 虚拟机的管理

随着数量和复杂性的增加，管理虚拟机成了一项艰巨的任务。每种实现都有自己专有的定义和维护虚拟机的机制。Open Virtualization Format（OVF）是打包和分布虚拟机的公开标准，涉及很多关于创建和维护虚拟机的值得关注的项目。

12.3.1 Vagrant

虚拟机管理的挑战之一是，每种虚拟化技术在创建和维护环境时都有自己特定的流程、脚本和工具。在 VirtualBox、VMware、AWS 和其他服务提供商提供的虚拟化技术之上，Vagrant（http://www.vagrantup.com/）提供了创建和配置可复制、可移植的虚拟机的机制。一个 `vagrant` 命令即可完成所有相关操作。

`Vagrantfile` 文件包含了对特定机器的配置。一旦该文件就绪，就需要添加一个盒子（创建虚拟机的基础镜像）。这是一个通过克隆得到的初始镜像，从未被修改过。盒子添加后，可以使用 `vagrant up` 命令启动虚拟机，通过 `ssh` 使用 `vagrant ssh` 访问新创建的虚拟机。还有其他命令用来提供一种机器以停止和清除旧的虚拟机和盒子。Vagrant 的创始人 Mitchell Hashimoto（http://mitchellh.com/）编写了一本关于 Vagrant 的书（http://shop.oreilly.com/product/0636920026358.do），该书深入讲解了 Vagrant。他最近还开发了一个名为 Packer 的新项目，该项目简化了跨 VM 实现的配置。

12.3.2 Packer

虽然 Vagrant 很好用，但是创建和维护镜像仍然是一个烦琐、困难且大部分需要人工操作的过程。Packer（http://www.packer.io/）使用一种可移植的输入格式编写模板，从而可并行为多平台生成镜像。Packer 为各虚拟机提供商自动生成基础镜像。Packer 中的创建者组件以构件的形式为指定平台创建虚拟机镜像。

举个例子，Packer 中的 VirtualBox 构件能创建 VirtualBox 虚拟机，并将其以 OVF 格式导出。构件由 ID 或者表示虚拟机镜像的文件组成。Packer 呼应了 Vagrant 的功能，使用 post-processors 可以获取构件并将其转换成 Vagrant 盒子。

使用一致的语法和工作流为不同提供商配置虚拟机的做法并没有考虑提供和维护虚拟机，使用一些 shell 脚本能初步加入一些自动化。针对 OS X 的 csshX（https://code.google.com/p/csshx）增强了控制台能力，可以对多台机器运行 ssh 命令，还有像 Capistrano（http://capistranorb.com/）这样的工具，在小型环境中管理多个服务器是够用的，但是当服务器数量增加时，这些方案对于一般性的系统管理任务就捉襟见肘了。

12.3.3 DevOps配置管理

可以通过命令行或 Vagrantfile 文件中引用的 Shell 脚本创建简单的 Vagrant 机器，更复杂的配置则需要使用 Chef（http://www.getchef.com/chef/）或 Puppet（https://docs.puppetlabs.com/）自动化安装和配置虚拟机。Puppet 和 Chef 都是用 Ruby 编写的，不过 Puppet 使用一种基于 JSON 的语言，根据依赖描述决定安装什么，而 Chef 仅需要使用 Ruby 来编写安装脚本。最近又出现了 Ansible（https://github.com/ansible/ansible）和 Salt（http://www.saltstack.com/），它们提供了其他可选方案（或者说补充方案）。这些工具在功能上存在大量的重合，但是每件工具都各有所长，适合特定的项目和管理员。

表 12-1 列出了 DevOps 配置管理工具。

表12-1：DevOps配置管理工具

工具	初始发布日期	说明
CFEngine（http://cfengine.com/community/download/）	1993	基于 C 语言，快速、轻量级，学习曲线陡峭
Capistrano（http://capistranorb.com/）	2005	主要针对 Rails 应用的部署
Puppet（https://docs.puppetlabs.com/）	2005	灵感源自 CFEngine
Chef（http://www.getchef.com/chef/）	2009	使用 Ruby 配置
Salt（http://www.saltstack.com/）	2011	快速、大型系统的协调和管理
Ansible（https://github.com/ansible/ansible）	2012	简单、不需要代理即可配置

标注出初始发布日期有助于读者理解工具的角色以及和已有工具之间的关系。Puppet 受到了 CFEngine（http://cfengine.com/product/what-is-cfengine/#puppet）的启发，Chef 的作者从 Puppet 的使用中学到了很多（http://docs.getchef.com/chef_why.html），但是在管理体验上采取了不同的方式。最近，与流线型工具 Chef 及 Puppet 类似的 Ansible（https://github.com/ansible/ansible）和 Salt（http://www.saltstack.com/）因其简单和平滑性，也变得流行起来。它们都能够初始化服务器配置，并且执行命令从任意节点上获取结果。

DevOps

科技界存在这样一种常见现象，当工具、技术和缩写在某一领域出现的次数达到一定程度后，就会出现一个对应的工作职位。DevOps 是 2009 年出现的一个概念，它代表了一种职业角色，其职责涵盖传统的开发和运营工作。开发者个体可能不会深入使用本节介绍的工具，但是了解这些工具有助于和专业 DevOps 工程师进行有效交流。

12.4 容器

全虚拟化是很多实用应用的早期目标，但是很多有限的虚拟化形式也很有用。部分虚拟化只模拟整个操作系统的一部分，不提供完整的虚拟机。操作系统专有的容器技术就是一种

有限的虚拟化形式。

进程隔离和系统安全问题驱动了容器技术的发展，这些无法通过操作系统本身的机制实现。传统的用户群组管理很麻烦，而且对很多情况来说不够完善。20 世纪 70 年代末，chroot 实用工具实现了有限的隔离，尽管在有些情况下很有用，但它欠缺运行一个独立的、功能完整的容器的能力。从上层看，容器是一个部分的虚拟机，从底层看，容器是一个增强版的 chroot。

人们将容器描述为*操作系统级的虚拟化*，当然，各厂商自己还有其他的叫法。它在容器中提供拥有专用资源的用户空间，但是所有命令均在宿主机器内核中运行。和模拟整个*机器*不同，容器技术专注于在服务器上虚拟化彼此独立、安全的*操作系统进程*。

12.4.1 LXC

LinuX Container（LXC，https://linuxcontainers.org/）虚拟化可用于 Linux 系统，它允许一台服务器上运行多个彼此独立的容器。和在独占系统上运行标准的操作系统级进程相比，它在资源利用和安全性之间做出了更好的平衡。容器直接运行 CPU 的原生指令，而不必像标准的虚拟化技术那样需要通过中间步骤，因此具有更好的性能。由于没有完整操作系统所需的开销，因此容器更加轻量级，相比一个完整的虚拟机，会占用更少的资源。Linux 容器在各版本的内核和发行版上皆可运行。

12.4.2 Docker

Docker（https://www.docker.com/）扩展了 LXC，提供了更高级别的 API。和其他容器技术类似，Docker 试图简化应用的打包和部署，为终端用户创建彼此独立的私有环境。从很大程度上说，Docker 让 LXC 的功能使用起来更为简单。从这个角度看，Docker 之于 LXC 就像 Vagrant 之于其支持的底层虚拟机实现，如图 12-2 所示。

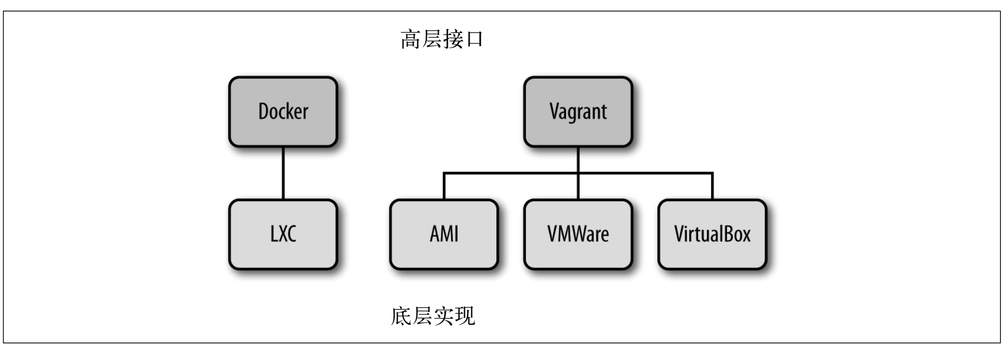


图 12-2：Docker 和 Vagrant 的高层 API

用 Docker 的术语来说，运行的*容器*基于*镜像*。对于给定容器，其文件系统状态和退出值

被保存下来，但是不保存内存状态。容器可以被启动、停止和重启。容器也可通过 Docker 的 `commit` 命令导出为一个镜像，这个镜像随后可以被用作新容器的蓝本。

Docker 中的镜像可以有父镜像，但基础镜像没有父镜像。用来创建容器的镜像保存于 Docker 库中，Docker 库被注册到一个注册器里，`index.docker.io` 是隐含的顶级注册器，因此 Docker 包含了发布和共享镜像的机制。

虽然这是一个新项目，但 Docker 潜力巨大，它确保能在分布式环境中提供标准的容器。如果使用合理，就能省去开发者和系统管理员安装机器的时间。Linux 系统在各种硬件上的流行为应用的发布提供了新的可能，应用可以部署在开发者的机器、云服务和嵌入式设备上。

12.5　项目

本章的项目用到了前面提及的多种虚拟化技术。项目需要提前安装 Git、VirtualBox（https://www.virtualbox.org/）和 Vagrant（http://www.vagrantup.com/）。只需几个步骤就可以设置好 Docker（在一个由 Vagrant 管理的 VirtualBox 上）。图 12-3 展示了运行于 OS X 系统之上的 Vagrant 文件，定义了一个 VirtualBox，Docker 实例运行在该虚拟机之上。Docker 容器里安装一个 Java SDK，然后就可以在容器里编译和运行 Java 程序了。这个例子虽然简单，但却包括了安装更大、更复杂的 Docker 容器所需的全部步骤。

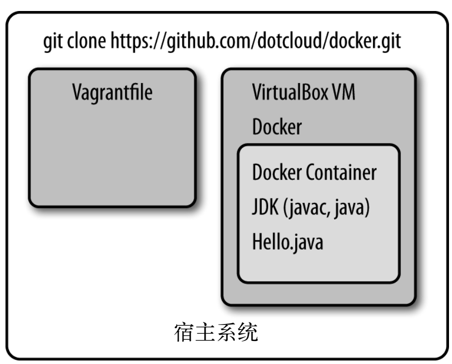


图 12-3：让 Java 运行在 OS X 系统上的 Docker

使用 Vagrant 安装 Docker（http://docs.docker.com/installation/）的方式因操作系统而异，总体上来包括以下步骤：

(1) 使用 Git 抓取安装机器所需的包含 Vagrantfile 的 Docker 源码；
(2) 使用 Vagrant 安装虚拟机；
(3) 用户通过 `ssh` 登录新建的虚拟机，切换到 Docker 用户；

(4) Docker 就绪后，就可以用来创建和维护容器了。

从宿主机器上可以使用 Git 下载 Docker 项目：

```
git clone https://github.com/dotcloud/docker.git
cd docker
```

Docker 正在紧锣密鼓地进行开发，变化非常快。初始化安装后，你可以使用 Git 更新 Docker：

```
git pull
```

Docker 为 OS X 系统提供了一个受 Vagrant 管理的 VirtualBox 虚拟机。使用如下命令启动和登录持有 Docker 的虚拟机：

```
vagrant up
vagrant ssh
```

登录成功后，键入 `exit` 可以随时退出 Vagrant 虚拟机。从 Vagrant 管理的盒子里，使用如下命令调用 Docker：

```
sudo docker
```

使用如下命令下载并安装一个基础的 Ubuntu 镜像和一些标准的 Linux 工具：

```
sudo docker pull ubuntu
sudo docker run ubuntu /bin/echo Docker is running!
```

12.5.1 Docker帮助

你可以使用内置的帮助系统学习 Docker。由于项目变化很快，在线文档可能已经不适用于你当前使用的版本。键入 `docker` 列出可用命令，添加 `-help` 参数列出每个命令的可选项：

```
docker
docker build -help
```

12.5.2 镜像和容器的维护

使用 Docker 工作一段时间后，会积攒下一些镜像和容器。使用 `info` 命令给出系统信息报告，其中包括容器和镜像的数量：

```
docker info
```

报告中包括了总数、已退出的容器和中间镜像，人们通常对这些东西感兴趣。容器退出后依然存在，直到手动将其删除。镜像可以迅速累积，因为它们创建于 Docker 文件中定义的每一步。`ps` 命令列出了运行的 Docker 容器，`images` 命令列出了镜像（不包括那些用于

构建的中间镜像）：

```
docker ps
docker images
```

使用 `-a` 列出所有的容器和镜像，加入 `-viz`，使用 GraphViz（http://www.graphviz.org/）可查看镜像的 .dot 图：

```
docker images -viz > docker1.dot
```

使用 `docker inspect <container name>` 可查看更为详细的容器信息。使用 `rm` 命令清理容器和镜像。命令可以串行执行：

```
docker rm $(docker ps -a -q)
docker rmi $(docker images -q)
```

12.5.3 在Docker里使用Java

Docker 的 Git 库包含了一个 Vagrantfile 文件，Vagrant 使用该文件建立和配置虚拟机。Docker 使用 Dockerfile 建立和配置容器。`FROM` 指令表示新虚拟机的基础镜像。有一些公共镜像库可供使用，或者你也可以选择使用自己的镜像。`MAINTAINER` 指明了镜像作者，`RUN` 指令在当前镜像上运行命令并返回结果。下面几步安装了 Oracle 的 Java 7 SDK，并接受其授权方式。`ADD` 指令将 Hello.java 复制到容器中，它将在那里被编译和执行：

```
FROM ubuntu:precise
MAINTAINER Casimir Saternos

RUN echo "deb http://ppa.launchpad.net/webupd8team/java/ubuntu precise main"\
| tee -a /etc/apt/sources.list
RUN echo "deb-src http://ppa.launchpad.net/webupd8team/java/ubuntu precise main"\
| tee -a /etc/apt/sources.list
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys EEA14886
RUN apt-get update

RUN echo oracle-java7-installer shared/accepted-oracle-license-v1-1 select true\
| /usr/bin/debconf-set-selections
RUN apt-get -y install oracle-java7-installer
RUN update-alternatives --display java
RUN echo "JAVA_HOME=/usr/lib/jvm/java-7-oracle" >> /etc/environment
ADD Hello.java Hello.java
RUN javac Hello.java
```

在 Dockerfile 所在目录创建一个简单的 Hello World 程序：

```
public class Hello{
  public static void main (String args[]){
    System.out.println("hey there from java");
  }
}
```

有了 Dockerfile 和 Java 类，就能从镜像开始构件容器了。-t 选项指定了库名，在列出可用镜像时会看到：

```
docker build -t cs/jdk7 .
```

另外，前面安装过的 JDK 也安装好了，使用如下命令执行复制到 Docker 容器中的程序：

```
docker run cs/jdk7 java -version
docker run cs/jdk7 java Hello
```

需要说明的是，这些命令是在 Docker 容器里执行的。在 Vagrant 虚拟机里执行会得到不同的结果（显示虚拟机中未安装 Java）：

```
java -version
java Hello
```

通过为上述 Dockerfile 增加新的定义，可以配置出能以 WAR 形式运行在 Jetty 上的 Web 应用。使用 ADD 命令可以将文件从 Vagrant 虚拟机复制到 Docker 容器，使用 RUN 命令能运行 wget 或其他实用工具从指定 URL[2] 下载需要的文件：

```
ADD rest-jersey-server.war rest-jersey-server.war
RUN wget http://repo2.maven.org/[……真实路径见脚注……]/jetty-runner.jar
```

将文件复制到 Vagrant

你或许想知道 rest-jersey-server.war 是怎样到 Vagrant 虚拟机上的，或者聪明如你，已经使用 Curl 或者 wget 下载了它。从网上下载是个不错的选择，但通过文件共享或使用 scp 也能复制文件。

默认情况下，Vagrant 共享了 Vagrantfile 的所在目录到虚拟机中的 /vagrant 目录。此外，Vagrant 默认使用 SSH 协议（从端口 22 到 2222），因此可以在宿主机器和虚拟机之间使用 scp 复制文件。比如，使用如下命令将宿主机器上的 docker.png 复制到虚拟机，弹出需要密码的提示时输入 vagrant：

```
scp -P2222 vagrant@localhost:docker.png .
```

然后就可以从镜像构件一个容器，并且使用它了：

```
docker build -t cas/restwar .

docker run -p 49005:49005 -name restwarcontainer cas/restwar \
java -jar jetty-runner-8.1.9.v20130131.jar \
--port 49005 rest-jersey-server.war
```

注 2：该 URL 路径太长，不支持换行，其地址是 http://repo2.maven.org/maven2/org/mortbay/jetty/jetty-runner/8.1.9.v20130131/jetty-runner-8.1.9.v20130131.jar。

需要注意的是，虽然容器在交互运行，但还可以继续打开其他 `vagrant ssh` 会话执行其他命令。如果你不需要以交互方式运行 Web 应用，那么启动应用服务器的命令需要是 Dockerfile 中的最后一个 `RUN` 指令。

默认情况下，Docker 会为新启动的容器起一个名字。使用 `-name` 选项可以为容器起一个有意义的名字，但这就额外引入了需要人工干预的步骤。如果需要重新运行上述命令启动容器，则必须重新命名容器或删掉之前创建的容器：

```
ID=$(docker ps -a | grep restwar | awk '{print $1}')
docker rm $ID
```

12.5.4 Docker和Vagrant的网络设置

在 OS X 或 Windows 上使用 Docker 的困惑之一是，它包括一台机器和两层虚拟化，如表 12-2 所示。Vagrant 安装于物理机之上，提供了一个完整的虚拟机，Docker 容器运行于该虚拟机之上。从不同位置可以看到不同的 IP 地址，有一些端口必须打开。

表12-2：项目服务器

服务器	描述
宿主机器	包括 VirtualBox 软件，使用 Vagrant 管理
Vagrant 实例	包含一个 VirtualBox Linux 虚拟机，上面安装有 Docker 软件
Docker 实例	安装了 Jersey 服务器，Web 应用运行其上

Vagrant 需要连接宿主机器上的一个端口，通过该端口配置 Vagrant 文件，运行 Docker 实例的命令也需要通过该端口。从外面来看，宿主机器只监听和响应该端口。网络设置的可能性很多，该例子只需要几步就可设置完成。

首先，我们在 Vagrant 上打开一个端口，这样宿主机器就能看到上面运行的是什么。在虚拟机上指定一个端口和宿主机器上的一个端口共享，这称为*端口转发*，端口号可以和宿主机器上的端口号相同或不同。在这个例子中，我们通过修改 Docker 提供的 Vagrantfile 将 Vagrant 虚拟机上的端口 49005 转发至宿主机器上的相同端口：

```
...
Vagrant::Config.run do |config|
  ...
  config.vm.forward_port 49005, 49005
  ...
```

在该容器中运行前面列出的 Jetty 中的 WAR，可通过在 Vagrant 虚拟机和访问页面上执行一些命令获取 ID 和 IP 地址：

```
ID=$(docker ps | awk '{print $1}' | grep -v CONTAINER)
IP_ADDRESS=$(docker inspect $ID | grep IPAddress | awk -F'"' '{print $4}')
```

```
echo $ID
echo $IP_ADDRESS
curl $IP_ADDRESS
```

这里的 IP 地址只在 Vagrant 虚拟机内部有效，外面是看不到的。这时，在 Vagrantfile 中指定的端口转发就起作用了。在宿主机器上，通过浏览器访问 http://localhost:49005/ 就能看到 WAR 的主页。

12.6 小结

20 世纪 80 年代，“虚拟现实”因 Jaron Lanier 变得流行起来。从那以后，电影、视频游戏和复杂的仿真系统都受益于虚拟现实的发展，对现实世界的虚拟化也对计算机从业者产生了深刻的影响，他们开始考虑如何将计算机硬件虚拟化。

Java 的成功很大程度上归根于 Java 虚拟机——一个屏蔽了底层操作系统细节的抽象层。Servlet 容器和 JEE 应用服务器更进一步，提供了更高级别的抽象。更高级别的抽象让专业化程度更高，这就抹去了中间层的很多问题。使用现代化的虚拟技术能让客户端 - 服务器端应用轻松运行于高扩展性的方案之上，而且因其结构化、隔离化的架构，进行分别打包和部署也很容易。虽然不能完全等同，但那些可以提供和底层系统类似功能的技术存在着明显的优势。

引用一个虚构角色的话来为本章开篇再合适不过了。电影中一个引人注目的角色——无论多讨人喜欢，总归和真人是不同的。即便如此，虚拟化还是能以某些形式模拟底层技术，从某些角度看起来，和它模拟的物理世界别无二致。

第13章

测试和文档

“没有语言，思想就像一处模糊、人迹罕至的星云。在语言出现之前，没有什么是清楚的。”

——索绪尔

詹姆斯·林德是一位 18 世纪的苏格兰医生。在英国皇家海军服役时，他完成了一项实验，该实验现在被认为是历史上第一次药物临床试验。他将 12 名生病的水手分成两组，分别提供不同的食物和治疗方式，他发现柑橘和柠檬能有效预防坏血病。现在我们知道坏血病是由于缺乏维生素 C 造成的，这中间整整经历了一个世纪，人们做着类似的实验，直到 1912 年，卡西米尔·冯克才提出“维他命”一词。

临床试验是科学方法中的一步。牛津词典（http://www.oxforddictionaries.com/us/definition/american_english/scientific-method?q=scientific+method）这样定义科学方法：这是自 17 世纪就奠定了自然科学基础的一种流程，包括系统的观察、测量和实验，然后形成假设，测试并修改它。这个定义虽然准确，但是却没有抓住重复试验最重要的输出，将对于现象的描述和假设以清晰、明确的语言表达出来。测试最好能得出对所测对象清晰、简洁的描述，这样有利于在后续研究中清楚地交流。

软件测试也有类似的历史渊源，它采用了过去四百年里自然科学中一直沿用的流程。测试是用来证明或推翻假设的。拿软件测试来说，就是假设全部或部分系统功能按照描述的方式工作。软件测试还能发现如何更好地设计应用，使应用更加模块化、结构化。另外，软件测试还有助于明确需求，用精确的语言描述待测试系统。

13.1　测试的种类

大多数软件开发项目都宣称是经过测试的，但这究竟意味着什么恐怕就不那么显而易见了。测试是一个宽泛的概念，根据测试结果、测试创建方式、整个系统的测试百分比以及参与测试的人员角色，可将测试划分为不同的种类。

13.1.1　“正式”与“非正式”

当测试目的不是很明确，也不要求测试结果非常正式时，就可采用*随机测试*。这和在 REPL 里练习编程类似。它为其他测试热身，也是一种开始接触不熟悉应用的方式。正式的*验收测试*则与此相反，它结构化、有组织、定义清晰，专门用于决定客户是否应该接受该系统。非正式测试通常是手动测试，正式测试则是高度自动化的。

与记住一堆测试种类和方法论相比，更重要的是理解测试目的，在无需引入不必要开销的情况下选择一种对项目来说足够严谨的测试方法。如图 13-1 所示，测试的正式程度可以用一根连续的箭头描述，这中间有各种可能性。虽然本章提到了科学方法，但项目测试方式的选择更像是一门艺术，而非科学。

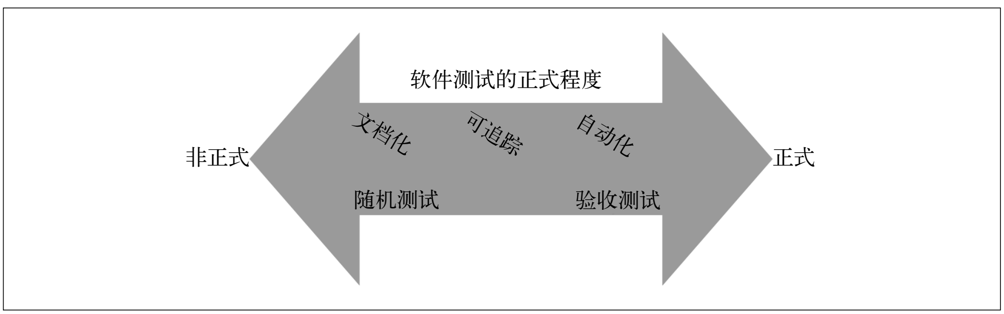


图 13-1：测试的正式程度

13.1.2　测试范围

测试范围差异很大。简单的*冒烟测试*（或者*健全测试*）仅仅能确保系统没有明显的破坏性缺陷，而*穷尽测试*则可以测试系统的方方面面。可以从不同角度量化系统的测试范围。*测试覆盖率分析*给出了代码的哪些行被单元测试跑到了。即使系统的测试覆盖率是 100%，依然需要穷尽所有的输入组合方能说这是一个穷尽测试。很少有项目能做到这点，即使那些宣称做到的或许也只是针对系统的功能测试。

*非功能性*的属性也应测试，包括负载测试、压力测试、多平台、多浏览器、多设备测试。云部署扩大了测试的边界，包括主动引入系统失败来保证系统的可恢复能力。比如 Netflix 公司的 Chaos Monkey 项目（http://techblog.netflix.com/2012/07/chaos-monkey-released-into-

wild.html）就运行在 Amazon Web 服务之上，其通过终结自动伸缩群中的虚拟机实例来检验系统是否能承受服务器的失效。

13.1.3 谁来测？测什么？为谁测

测试的时间、方式、创建者和受众都是要重点考虑的内容。开发人员在编写代码之前就编写好了单元测试，或者在编写好代码后再添加单元测试。黑盒测试通常需要测试人员编写更高级别的功能测试。单元测试和实现代码紧密相关，而黑盒测试则不需要考虑软件的内部结构。高度的独立测试或许能验证系统的某一细微方面，而集成测试则可以保证相关系统的接口之间没有缺陷。

也可以从谁创建测试并评估结果的角度来看待测试，开发者首先看到测试结果，然后是 QA 分析师，最终到达决定是否接受这个系统的决策者手中。测试可由不同的人编写，包括测试人员和开发者。测试是根据需求创建的，因此需要最大限度地让定义需求的业务分析师和业务干系人参与进来。事实上，定义好测试有助于开发一套通用的、没有歧义的语言，便于系统测试人员进行交流。

13.2 测试反映了组织的成熟度

在实践中，对软件系统的需求可能来自多处并以多种形式存在。虽然文档被认为是需求的主要来源，但不是所有的项目都有正式的文档。即使项目有正式的需求定义和系统描述，在一个处于开发活跃期的系统里，如果不是自动生成或有意维护，这些文档也很快就会过期。程序员的记忆和代码中的注释描述了系统期望的行为，没有这些，可能就只有代码能用来描述系统功能了。人们之所以认为测试是系统之外最能精确定义软件需求的，无法获取代码是主要原因。

事实上，一组定义良好的测试，即使在实现系统的原始代码被取代后，依然能够工作。从这个角度来看，测试比在建系统本身更长久、更有价值。它们是在组织之间、干系人之间沟通系统状态和功能的高效方式。

13.2.1 使用软件能力成熟度模型评价流程

Conway 法则（http://www.melconway.com/Home/Conways_Law.html）表明：设计系统的组织，其设计结果都是对所在组织交流结构的复制。也可以这样说，一个组织的软件测试状态反映了软件流程的成熟度（从软件流程的定义到控制）。软件能力成熟度模型（Capability Maturity Model，CMM，https://resources.sei.cmu.edu/asset_files/TechnicalReport/1993_005_001_16211.pdf）定义了各阶段的成熟度结构，用以描述软件开发和维护流程中反映出的工艺稳定性和组织纪律性，如表 13-1 所示。

表13-1：软件能力成熟度模型的各个阶段

级别	名称	描述	测试
级别 1	初始级	不一致、无组织	没有测试或进行随意测试
级别 2	可重复	规范的流程	有一些单元测试
级别 3	已定义	标准的、一致的流程	统一的项目，每次构建都运行单元测试
级别 4	已管理	可预测的流程	持续集成测试
级别 5	优化中	不断改进流程	覆盖率、代码质量、报告

高度优化的组织对团队中英雄人物的依赖度更低，更不容易错过最终日期、“死亡之旅”（http://en.wikipedia.org/wiki/Death_march_(project_management)）和其他代表项目失去控制的现象。关注应用的扩展性，就有可能忽略组织的扩展性。

如果想让组织获得长期发展，则需要更高级别的软件成熟度模型。统一的流程和程序能让新成员更快地融入组织。生产效率高的成员有时要放下工作，花时间教授新成员一些内部的、未文档化且非强制执行的惯常做法，而统一的流程能将这一时间最小化。统一的流程还能减少新员工有可能给现有系统带来的不稳定性。在最佳情况下，好的流程能教会员工一些方法，这些方法同样适用于其他尚未被充分掌控的项目。

13.2.2　使用Maven促进流程统一

直接说服人们采用统一的软件开发实践很难，而使用工具推动实践向组织目标靠拢就容易得多。比如，Maven 引入了一致的构建周期，它通过插件提供了扩展的灵活性，且不至于和很多最佳实践偏离太多。

Maven 的目标之一是提供统一的构建系统（http://maven.apache.org/what-is-maven.html）。和 Gradle 或者其他构建工具相比，Maven 可能不够灵活。统一性或多或少会让软件有点僵硬，但不能简单地说僵硬就是邪恶的，应该避免的。必须做出决定，实现那些有利于组织内部流程统一的结构和工具。结构化的测试和报告流程是最基本的，适用于大多数软件项目。

Maven 默认包含两阶段测试（测试和集成测试）。如图 13-2 所示，测试位于 Maven 其他构建环节的中间。通常情况下，测试阶段会使用 Surefire 插件（http://maven.apache.org/surefire/maven-surefire-plugin/），集成测试阶段则使用 Failsafe 插件（http://maven.apache.org/surefire/maven-failsafe-plugin/）。可使用 Surefire Report 插件（http://maven.apache.org/surefire/maven-surefire-report-plugin/）以 HTML 格式显示 Surefire 和集成测试报告。

Maven 的构建生命周期是推动项目统一化的方式之一，默认包含单元测试和集成测试两个阶段。它们处于流程的中间，生成的文档可发布到统一的地方（Maven 网站）。默认阶段提供了一个良好的基本框架，如有需要，可在此基础之上扩展 Maven，处理其他测试情况。

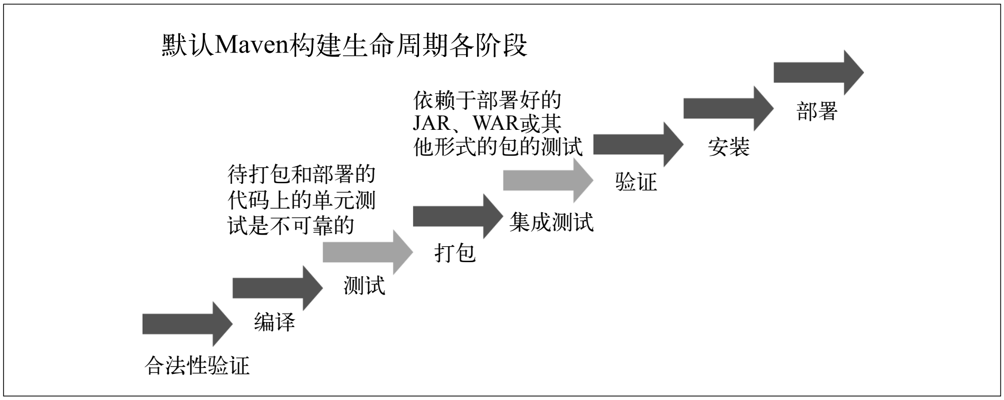


图 13-2：Maven 构建生命周期

Maven 可在持续集成（Continuous Integration，CI）服务器上周期性地执行应用的单元测试，作为构建流程的一部分。这样的服务器很多，如开源的 Jenkins（http://jenkins-ci.org/），商用软件 Team City（http://www.jetbrains.com/teamcity/）和 Bamboo（https://www.atlassian.com/software/bamboo）。持续集成服务器提倡标准的软件开发最佳实践。持续集成服务器要求使用版本控制系统，鼓励合理维护 Maven 资产，并能够快速反映出构建能否在合理时间内完成。

持续集成服务器可将 Maven 报告（http://maven.apache.org/plugins/maven-source-plugin/project-reports.html）发布到指定地址。这些报告展示了代码的结构和质量，同时也让测试结果一目了然，包括编写了多少条测试、执行结果、性能和总体上的代码覆盖率。



避免追求 100% 的覆盖率

人们一旦使用覆盖率报告，就很容易为追求 100% 的测试覆盖率而忽视测试本身的价值。覆盖率工具覆盖了那些会被执行的代码路径。现实中，有些代码路径永远不会执行，还有一些则根本不值一测。因此测试*覆盖率*不等于测试*质量*。究其本质，测试只能证明有问题，不能证明没问题。过份强调测试覆盖率会让程序员做出曲意逢迎的改动，而软件质量则没有任何提高。

JUnit 的制定者意识到人们有过份追求测试覆盖率的趋势，也知道有些代码根本就不需要测试。他们在 FAQ（http://junit.sourceforge.net/doc/faq/faq.htm#tests_6）中有句座右铭：

“当担心变为麻烦时再进行测试。”

一个组织可通过使用 Maven 将单元测试集成到所有项目中并将项目构建都放在持续集成服务器上的方式，迅速到达软件能力成熟度模型 3 级。发布测试结果有助于开发团队之间沟通软件状态和相关测试情况。虽然 Maven 支持手写文档，但是并没有涵盖将需求和测试相

关联的机制。

通过持续记录项目统计信息，可以不断地改善流程、更好地审视项目。和代码复杂性、测试覆盖率、代码行数、重复代码、注释和违反编程实践相关的数据是客观可度量的，可使用类似 SonarQube（http://www.sonarqube.org/）这样的产品，将这些信息保存到数据库中。这有助于识别代码质量的变化趋势，是逐渐改善还是不断退化。

13.2.3 使用行为驱动开发促进流程统一

非正规测试通常是为了找出缺陷或使系统失灵。这样的测试需要保留，经常性地运行这样的测试可保证问题不在日后重现。构建系统时经常运行的一组单元测试能为应用的后续支持提供非常高效的安全保障。但是不应孤立地看待测试，而应将它们和软件开发的其他方面融为一体。可参考 1993 年 Steve McConnell 在《代码大全》（*Code Complete*，英文版由 Microsoft Press 出版，http://www.cc2e.com/Default.aspx）一书中的话："在写方法时，想想该如何测试。在你编写单元测试，或是测试人员独立测试你的方法时，这样的想法都是有益的。"

值得注意的是：*边设计边测试*会带来更好、更结构化的代码，更好的命名和其他的好处。它表明测试和软件开发生命周期的其他阶段关系紧密，它还预测了最近的软件测试实践。

测试驱动开发（Test-Driven Development，TDD）采取了编写代码时考虑测试的想法，事实上，是在编写相关代码前先编写测试。具体来说，TDD 遵循这样一个循环：先写一个失败的测试来描述新需求，然后编码、重构，直到测试成功。

随后，Dan North 引入了*行为驱动开发*（Behavior-Driven Development，BDD），帮助开发者更好地理解该从何处开始 TDD：测什么？对于给定测试应该输入什么？如何命名？BDD 以一种描述性的方式架起了代码和人类语言之间的桥梁。TDD 给测试的创建方式以更开放的空间，而 BDD 沿袭了 TDD 的工作流，但是提供了更正规的格式用以描述行为。

BDD 的成功需要组织中的个人勇于承担，同时它也有一些方面和较高的 CMM 水平相关。BDD 要求团队之间的交流。质量保证测试人员、开发者和商业分析师必须高效协作。带来的好处是相关测试用例直接和需求联系在一起，并且使用一种通用、无歧义的语言进行表达。

使用 Maven 和一些成熟的测试框架，可以为开发者引入了一致的组织和结构。BDD 鼓励开发者进行交流并持续改进流程，而不仅限于测试和需求的收集定义。

13.3 测试框架

测试方式由测试软件决定。JUnit（http://junit.org/）用来做 Java 的单元测试，而 JavaScript 的单元测试使用 Jasmine（http://pivotal.github.io/jasmine/）。JBehave（http://jbehave.org/）是 Java 的行为驱动开发框架，而 Cucumber（http://cukes.info/）则是 Ruby 的行为驱动开发框架。

13.3.1 JUnit

JUnit（http://junit.org/）最初由 Erich Gamma 和 Kent Beck 编写，它发展迅速，从最开始的简单框架发展到了已经相当复杂的 JUnit4。在以前的版本中，单元测试归属特定的对象层次，方法名也必须遵守命名规范以便能正常工作。像 Java 生态系统中的很多项目一样，JUnit4 使用了大量注释。比如，不显式创建 `setUp()` 和 `tearDown()` 此类方法，而是使用 `@before` 和 `@after` 注释去标注这些在测试前后需要执行的代码。`@beforeClass` 和 `@afterClass` 用来标注在类初始化前和初始化后要执行的代码。`@Test` 替换掉了以前方法名以“test”开头的惯例，而 `@ignore` 取代了注释掉单元测试的讨厌做法。因为对象层次中已经没有 `assert` 方法了，所以需要导入那些静态断言类。断言还可用来标明带参数的测试或一组测试。

> **TestNG**
>
> 在过去几年中，另外一种引起人们注意的 Java 单元测试框架是 TestNG（http://testng.org/doc/index.html）。它和现在的 JUnit 功能类似，但以前它有一些 JUnit 没有的功能，比如带参数的测试。此外，它还能更好地管理一组测试。这为高效集成测试带来了灵活性。有些开发者喜欢 TestNG 的约定，同时也喜欢 JUnit 中的注释。

单元测试通常和主程序使用同样的编写语言。这样做的好处显而易见，那就是能轻松地执行具体的独立方法或应用的函数。随着单元测试的普及，它逐渐成为开发者之间用来沟通的“文档”。“这个项目是怎么工作的？”“看看单元测试就知道了。”遗憾的是，编程语言不能帮助那些非程序员清楚地理解系统是如何工作的。在很多情况下，即使掌握编程语言的程序员也无法从测试中获取需求。

Jasmine 和 Cucumber 用来实现 BDD，这在很大程度上支持了自然语言。Jasmine 使用 JavaScript 编写测试，而 Cucumber 从根本上说是一门名为 Gherkin 的小巧语言的解释器。Gherkin 是一种商业可读的 DSL，描述了软件的行为。实际用来测试应用的代码被单独写在另外的文件中。

13.3.2 Jasmine

Jasmine（http://pivotal.github.io/jasmine/）使用 JavaScript 字符串和函数来描述需求。它又小又轻，因此很容易和项目集成，只需打开浏览器就能运行。基本的测试结构包括对测试的描述，其后的块包含了期望，其实质是对返回结果是 `true` 还是 `false` 的断言。按惯例，这些代码存放在一个名为 Spec 的 JavaScript 文件中，打开 SpecRunner.html 运行测试，该页面包含相关的依赖类库：

```
describe("Suite Title Here", function() {
  it("Expectation (assertion) Title Here", function() {
    expect(true).toBe(true);
```

```
  });
});
```

虽然接近自然语言，但测试还是用 JavaScript 来编写，在那些没受过专业训练的人看来，就像是人类语言中参杂了大量的符号。外行可能会问："为什么最后两行末尾的符号看起来如此悲伤？"

13.3.3 Cucumber

Cucumber 基于一种名为 Gherkin 的 DSL，它和自然语言很像。比如 login.feature 是一个结构化的文本文件，里面有一些关键词和规整的对齐：

```
Feature: Login
  As a user,
  I want to be able to log in with correct credentials

  Scenario: Success login

    Given correct credentials
    When I load the page
    Then I should be able to log in
```

和每一步功能相对应的代码实现都是单独维护的：

```
...
Given(/^correct credentials$/) do
end

When(/^I load the page$/) do
  Capybara.visit(VALID_LOGIN_URL)
end

Then(/^I should be able to log in$/) do
  Capybara.page.has_content?('To NOT Do')
end
```

正则表达式用来匹配测试描述中的文本。Ruby 代码用来执行真正的测试。这个例子中，使用 Capybara 可自动打开浏览器，访问一个 URL，然后判断期望的内容是否出现。

摘自 The Cucumber Book[1]

Cucumber 有助于促进团队成员发现并使用一种通用语言进行交流，如此一来程序员和非程序员均可使用此语言。Cucumber 测试使用一种业务干系人能读懂的语言进行编写，同时可直接和开发者的代码交互。大家共同编写测试，分工协作，项目成员不仅要决定接下来需要实现的行为，而且学会使用一种大家都明白的通用语言来描述行为。

注 1：参见 https://pragprog.com/book/hwcuc/the-cucumber-book。

测试框架可以直接调用或通过构建工具进行调用。本章所示项目均可作为 Maven 构建的一部分运行。

13.4 项目

本章所示项目（https://github.com/java-javascript/client-server-web-apps/tree/master/Chapter-13-Testing/ToNotDoApp）展示了如何将 JUnit 和 Jasmine 测试集成进 Maven 构建。这其中还包括了 Cucumber 测试，使用 Capybara 调用 Selenium 来作为一个浏览器自动化功能测试的 BDD 例子。

为了避免实现一个常规的、落入俗套的“待办事项”应用，该项目包含一个“打死也不干”Web 应用，如图 13-3 所示。该应用可以用来列出人们不想干的事。亲爱的读者朋友们，你们是不是觉得我这个想法很新颖别致呢？

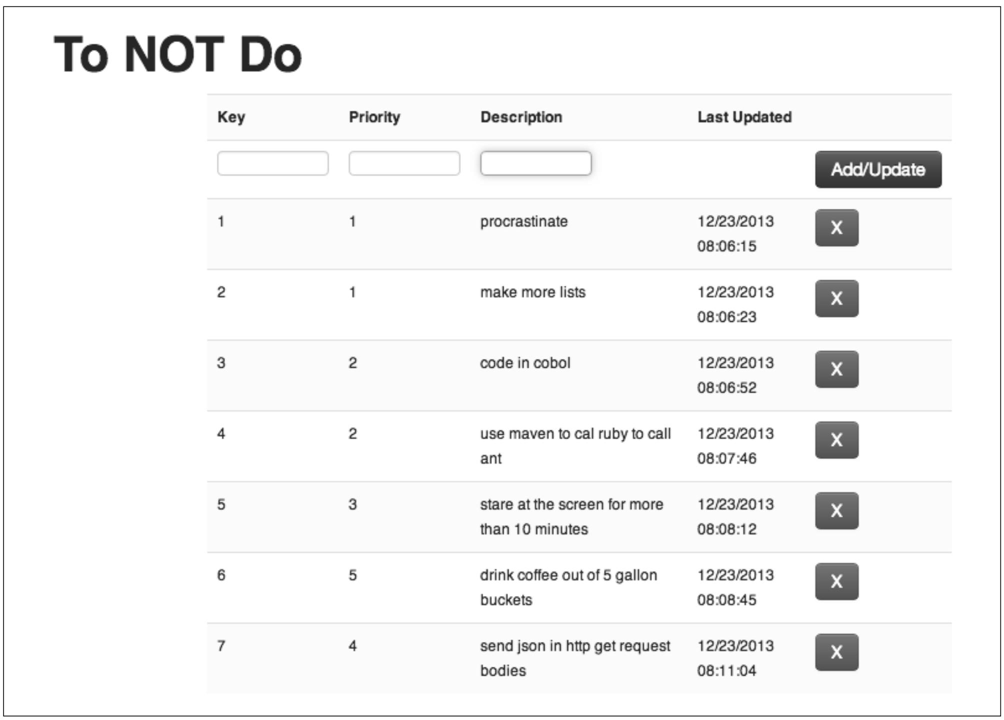


图 13-3：打死也不干 Web 应用

该项目展示了在 Maven 构建全过程中调用的各种测试框架。最终的结果是一个网站，包括了测试报告和相关文档。

13.4.1 JUnit

对于任何 Maven 项目，其配置的出发点都应是 pom.xml。依赖部分引入了 JUnit，它在测试时会用到：

```
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.11</version>
<scope>test</scope>
```

13.4.2 Jasmine

和在 pom.xml 中引入 JUnit 做 Java 单元测试的方式一样，需要引入另外的依赖来支持 JavaScript 测试。本项目使用了 Justin Searls 负责维护的 Jasmine Maven 插件：

```
<groupId>com.github.searls</groupId>
<artifactId>jasmine-maven-plugin</artifactId>
<version>1.3.1.3</version>
```

项目包含一个测试样例，用来验证所依赖类库是否存在，功能是否正常：

```
describe("Validate moment.min.js", function() {

    it("expects moment.min.js to be functional", function() {
                expect(
                    moment("20111031", "YYYYMMDD").
                    format('MMMM Do YYYY, h:mm:ss a')
                ).toBe(
                        "October 31st 2011, 12:00:00 am"
                );
        });
});
```

除了要在 Maven 中声明依赖，还需要额外配置引入 JavaScript 文件。如果配置正确，使用 `mvn install` 就能运行测试了：

```
...
[INFO]

J A S M I N E S P E C S

[INFO]
Suite Title Here
  Expectation (assertion) Title Here

Validate moment.min.js
  expects moment.min.js to be functional

Results: 2 specs, 0 failures
...
```

Jasmine 插件在很多情况下都是完美的解决方案。但是与浏览器相关的代码无法运行，这限制了插件适用性。浏览器自动化更适合需要大量 DOM 操作的测试。使用类似 Selenium（http://docs.seleniumhq.org/）的方案可在多浏览器上进行测试。浏览器自动化让浏览器的特性得以暴露出来。在 Java 单元测试中可以直接使用 Selenium，但很多情况下，浏览器自动化测试要求级别更高的功能或集成测试。使用类似 Capybara（http://jnicklas.github.io/capybara/）这样的高级类库作为 Selenium 的驱动类控制浏览器是非常高效的。下面这个 Cucumber 的例子（http://cukes.info/）就是这样使用 Capybara 的。

13.4.3 Cucumber

/ruby 目录包含一个 Ruby 版的登录测试。Gemfile 列出了依赖，webapp_steps.rb 包含处理功能文件中所定义的需求的代码。使用如下命令可启动应用：

```
mvn jetty:run
```

打开另一个会话，运行 Cucumber：

```
cd ruby
cucumber
```

列出的功能会逐个运行，输出显示了每一步的执行情况，如图 13-4 所示。

```
Cs-MacBook-Air:ruby cs$ cucumber
Feature: Login
  As a user,
  I want to be able to log in with correct credentials

  Scenario: Success login          # features/login.feature:5
    Given correct credentials      # features/step_definitions/webapp_steps.rb:18
    When I load the page           # features/step_definitions/webapp_steps.rb:22
    Then I should be able to log in # features/step_definitions/webapp_steps.rb:26

Feature: WebService
  As a user,
  I want to be able to log in, add and and list the feature through the API

  Scenario: Success add/list                               # features/webservice.feature:5
    Given correct credentials                              # features/step_definitions/webapp_steps.rb:18
    When I add and list To Not Do items                    # features/step_definitions/webapp_steps.rb:32
    Then the returned list should contain the item I added # features/step_definitions/webapp_steps.rb:37

2 scenarios (2 passed)
6 steps (6 passed)
0m3.912s
```

图 13-4：运行 Cucumber

13.4.4 Maven报告

本章的 pom.xml 文件比书中其他项目里的文件长很多。大量和核心应用无关的依赖和插件需要在测试时用到。而且，还有一些配置用于将信息发布到 Maven 项目网站。

使用 Maven 的网站插件生成一个网站，运行在 8080 端口：

```
mvn site:site
...
mvn site:run
```

Maven 的 /site/ 目录包含很多资源文件，可用来定制网站。在图 13-5 中，通过添加一个 site.xml 来定制“皮肤”，为网站提供了不同的风格。

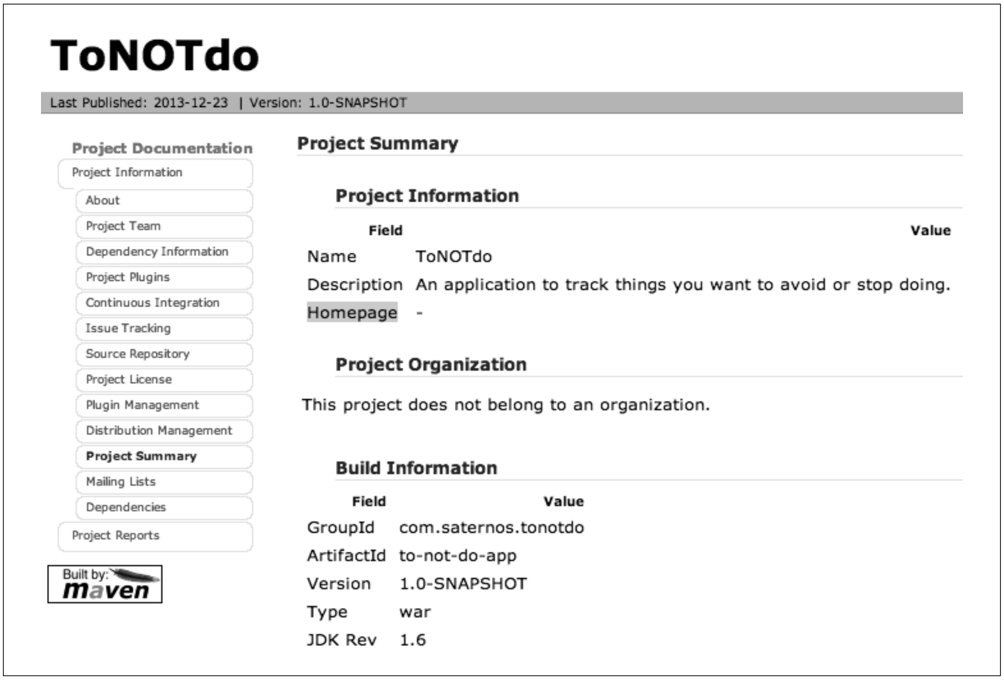


图 13-5：打死也不干 Maven 网站

13.5　小结

软件的可扩展性需要架构良好、可预测且高性能的系统，而测试保证了这种系统的开发。单元测试、Maven 和持续集成相关系统通过明确开发流程、掌握应用状态促进了这种系统的开发。

组织的可扩展性需要流程规范且沟通流畅。上述工具和 BDD 类的方法共同为团队带来了测试的益处，确保及时反馈软件的质量。

软件测试基于物理学中提炼出的科学方法和实践，这为软件提供了客观的支持。如果使用得当，测试为展示应用的质量和性能提供了实用价值，并促进在需求是如何实现的这一问题上达成共识。

第 14 章

总结

> “生命就是一个分布式系统。人与人之间通过一个分布式、超媒体系统进行交流，人类的智能、声音和手势、视觉和听觉，还有想象，都是其中的组件。”
>
> ——罗伊·T. 菲尔丁

最初打算写书时脑海里仅仅有个大概的计划，现在回头看看最终的结果，感觉这是一段令人着迷的经历。虽然和我的最初预想略有出入，但本书还是忠于了早先介绍的主题——变化。在第 1 章我们指出，由于技术的革新，世界发生了巨大的变化，这在很多领域是显而易见的。而且变化的速度在最近几年变得越来越快。一个大型项目，在其上线之日起就变成了一个遗留系统，这种事已经司空见惯了。

试图迎合所有的创新或者选择完全忽略它们，这两种极端反应都是目光短浅且徒劳无用的。更好的面对方式是找出那些真正改变游戏规则的因素。本书着重反映了一些 Java 开发者会碰到的问题，这需要他们更为深入地进行研究。有两种基本途径可应对持续改变的 Web 开发方式，即从更广大的开发社区和前人经验中进行学习。

14.1 社区

其他语言社区有着完全不同的观点，因此可以启发 Java 开发社区产生不同的创新。总体上来说，Java 开发者基础牢固，但有时候有点固步自封。他们能从在 JavaScript、Ruby 和 Python 等语言上成长起来的其他社区那里学到很多（同时也能贡献很多）。很多伟大的想法一经实现，便可应用于其他更为广泛的领域。一种环境下的新奇想法，可能在另一种环境下就成了革命性的概念。

14.2 历史

第二点是对过去的了解。当代文化对新事物总是不假思索地接受，实际上更好的做法是对创新保持开放的心态，从一个更广阔的视野上评估其价值。计算机科学和软件开发历史虽短，但内容却很丰富。创造了“面向对象”这一术语并发明了 Smalltalk 语言的计算机科学家艾伦·凯（Alan Kay，http://www.drdobbs.com/architecture-and-design/interview-with-alan-kay/240003442）曾经指出（http://www.bit.ly/1eUUzKE），现代计算机科学和软件开发的很多方面表现出和流行文化（http://ubm.io/1eQNBrD）相近的趋势，不关心也不了解过去。很多重要的、持久的想法其来有自，几年前甚至几十年前就已经存在了。如有可能，我们最好站在巨人的肩膀上，从过去所犯的错误中学习。

> “在过去 25 年左右的时间里，我们有点朝着流行文化的方向发展，就像电视发明之初，一些发明电视的人认为这是一种向大众普及莎士比亚的新方式。但是他们忘记了那些看电视的人首先要学养丰富，能看得懂莎士比亚。电视所能做的只是让人们看，至于能看出什么来就是他们的事了。
>
> 因此，我认为现在之所以缺乏真正的计算机科学和软件工程，部分要归咎于这种流行文化。”
>
> ——艾伦·凯

14.3 尾声

我希望读者能或多或少从本书中学到点什么。能在这样一个时代从事软件开发是件令人兴奋的事，时代的变化带来了前所未有的机遇。新技术层出不穷，永远有解决不完的问题。阅读本书，练习其中的项目能成为其中一步，希望这是你们中某些人的第一步，让您从中获益，并且创造出新的系统，让这个世界变得更加美好。我将引用伟大的数学家和老师乔治·波利亚的话给读者们留作思考。乔治·波利亚聪明过人，但他认识到人性和个人因素在解决问题中的作用，这对于软件开发同样适用。

> “认为解决问题是一种纯粹的“智力活动”是错误的，决心和情感在其中也起了重要作用。不温不火、懒洋洋地费些功夫也许足以解决一些课堂上遇到的常规问题，但是对于严肃的科学问题，唯有意志力才能引领你经受住长年累月的艰辛和挫折……教授如何解决问题，就是在教你如何培养自己的意志力。”
>
> ——乔治·波利亚

附录A

JRuby IRB及Java API

随着时间的推移，人机界面风格已经变化繁多，并且这种变化是基于有问题的设备特性以及看似有些随意的趋势。命令行界面（Command-Line Interfaces，CLI）曾是 20 世纪 90 年代之前与操作系统交互的主要手段，之后，它逐渐因为图形化操作系统以及可视化互联网的主导而黯然失色。尽管外观相对简陋，但 CLI 仍然深受欢迎，因为它提供的功能可能是 GUI 不具备的。

CLI 提供的交互模式类似于电传打印机（这是种从电报机演变而来的小玩意儿，用于发送键入的信息）的功能。CLI 比 GUI 更依赖于用户的打字能力。这种限制其实也是一种优势，因为 CLI 可以用于编写脚本。大多数的程序员都在他们所使用的操作系统中通过命令行 shell 的方式体验过 CLI。当涉及的任务越来越多时，它可以很轻易地将一组命令打包到一个脚本之中，这就导致在许多脚本语言中都包含各自的 CLI 作为执行环境。

也有人将 CLI 称为 REPL（读取 – 求值 – 输出循环，Read-Eval-Print Loop）或 language shell。无论你怎么称呼它，在执行给定的表达式或命令时，它对语言特性的探索及立即能得到反馈结果的特点都是非常宝贵的。

虽然 Java 本身不包含 CLI（Beanshell 与其最为接近；要详细了解 Beanshell，请参考 http://www.beanshell.org），但 JVM 所支持的语言，如 JRuby、Jython 及 Groovy 等都有自己的 CLI。JVM 和 Java 的紧密整合使得它可以通过所有这些语言的 CLI 与原生的 Java 类和模块进行交互。这种工作流上的显著差异在测试及调试时非常有用。在本章中，JRuby 的 IRB（交互式 Ruby shell，Interactive Ruby Shell）将通过 JDBC 连接到基于 Java 的数据库，并执行 SQL 查询。

> **为什么不叫 IRS?**
>
> 你可能会疑惑为什么 Ruby 不把自己的命令行界面称为 IRS（交互式 Ruby shell）。Ruby 程序的标准文件扩展名是 .rb。因此“交互”加上“RB”就是 IRB。

A.1　设置Gradle的使用

Gradle 可用于下载下面脚本用到的 Java 项目依赖。注意这只是出于便利，Gradle 这样的构建工具和 CLI 之间没有必然的联系。这个 Gradle 构建文件是用 `gradle setupBuild` 生成的。生成的 build.gradle 文件包含与使用相关的注释。后续会对这个文件进行修改以包含需要的模块及相关插件，还有仓库所在的主机位置：

```
apply plugin:'java'
apply plugin:'application'

repository{mavenCentral()}

dependencies{
    compile 'org.slf4j:slf4j-api:1.7.5'
    compile 'hsqldb:hsqldb:1.8.8.10'
    compile 'com.h2database:h2:1..3.172'
        compile 'net.sf.opencsv:opencsv:2.3'
        compile 'commons-io:commons-io:2.4'
        compile 'org.apache.derby:derby:10.10.1.1'
    testCompile "junit:junit:4.11"
}
```

定义了构建文件后，可以使用 `gradle build` 构建项目。所需的 JAR 包会被添加到本地仓库中。再次重申，这里的任务并不一定需要 Gradle，你可以使用 Maven 来替代，甚至可以手动定位和下载需要用到的 JAR 包。我们选择 Gradle 是因为它的语法很精简（与 Maven 相比），而手动下载文件很容易出错而且很烦琐。

A.2　JRuby IRB

因为下面的实例使用了 Java 类（数据库实现和 JDBC），所以需要一个基于 Java 的 Ruby 版本。其他的实现将无法使用。如果你正在使用 RVM，若是需要可以安装 JRuby 并且选择它来使用：

```
$rvm use jruby 1.7.4
```

如果你还没有安装过，那么先安装 gem 包（本例用的是版本 1.3.5）。当它可用之后，运行 `bundle init` 可创建包所需的 Gemfile。增加如下 3 行：

```
gem 'jdbc-derby', '10.9.1.0'
```

```
gem 'jdbc-h2', '1.3.170.1'
gem 'jdbc-hsqldb', '2.2.9.1'
```

这些 gem 包含了 JRuby 要用到的 JDBC。运行 `bundle` 来安装这些（Ruby）依赖。

我们已经见识了通过 Gradle 管理 Java 依赖，以及通过 bundler 管理 JRuby 依赖。在 JRuby 中直接引用 Java 的 JAR 也是可行的。方便起见，我们复制 JAR（在 Gradle 初始化时引入的）到当前路径下的一个 lib 目录。这个文件在你本地的 Gradle 库中：

```
$ find ~/.gradle -name opencsv-2.3.jar

$ mkdir lib

$ cp <path to jar file from find command>opencsv-2.3.jar lib
```

这个 JAR 在 JRuby 中可以使用 `require` 关键字来获取（这也是通常用来引入 Ruby 文件的）。

IRB介绍

登录 IRB 会打开一个提示符。具体的提示符会根据你当前使用的 Ruby 版本而变化：

```
$ irb
jruby-1.7.4 :001 >
```

在这个提示符后面，你可以输入一个表达式，并且立即看到执行结果：

```
2.0.0-p247 :001 > 1 + 4
 => 5
```

你还能检查对象，并找出它们提供了什么方法。Java 中的反射也允许观察和操作对象，但是略显冗长和复杂。作为对比，Ruby 提供了简洁明了的方式来动态查询和改变对象。在这方面的灵活性让它的元编程能力广为人知：

```
2.0.0-p247 :002 > "Hello World".class
 => String

2.0.0-p247 :014 > "Hello World".methods.grep(/sp/).sort
 => [:display, :inspect, :respond_to?, :split]

2.0.0-p247 :021 > "Hello World".split
 => ["Hello", "World"]
```

接下来的例子会忽略相关提示和运行结果。但是一个命令运行之后立即得到的反馈对于 `irb` 的交互具有巨大价值，谁用谁知道。

在前面的例子中，`Hello World` 是一个字符串（没有被指定到任何特定变量）。它有大量的方法可以调用，因此这个例子过滤（使用 `grep`）和整理了包含 `sp` 的方法。最终，以这种

方式得到的方法，我们可以称为对象（`Hello World` 字符串）上的方法，并且观察它的作用。这种方法并不是参考文档的替代品，但是确实可以省掉很多不必要的查找。

A.3　基于Java的关系型数据库

在一个 `irb` 会话中，我们现在可以探寻适用于基于 JAVA 的关系型数据库的 API。如果你使用过 JDBC 编写 Java 类，那么毫无疑问，你肯定记得这需要一定数量的模板文件。除了标准的 Java 规定（使用 `main` 方法来定义类），你需要编写大量的异常处理代码（`import` 语句，在方法声明中的 `throws`、`try/catch` 块）。这些额外的语法需要添加显式输入代码、少量输出和一些注释，并且一些看似简单的类会变得更加臃肿。幸运的是，Ruby 的语法比较简洁，并且代码编写的交互环境让测试驱动 API 变得简单。

这里我们简单看看 3 个数据库：H2、HSQLDB 和 Derby。从外部看来它们很类似，但是在具体实现、存储机制、性能和开源许可选项上都存在不同。每一个数据库都可以通过 JDBC 访问，在 `irb` 会话中我们会有效地测试每一种类型的 SQL 语句，如示例表 A-1 所示。

表A-1：SQL语句类型

类型	示例	描述
Query	SELECT	检索数据
DDL	CREATE、DROP	数据定义语言（创建、改变或替代一个数据库对象）
DML	INSERT、UPDATE、DELETE	数据操作语言（修改数据）

事实证明，只有你真正使用这些数据库时才能发现它们之间细微的区别（例如 SQL 语法、字符串连接和结果集的关闭）。

为了让这些例子更加简洁，我们需要一份可用于每一类数据库的通用代码，它可以添加到文件中并通过 `irb` 加载。这里的代码需要支持：使 Java 可用，能够装载 opencsvJAR 包（用于快速将一个结果集渲染成以逗号分隔的值），添加能够执行 SQL（查询、DML 和 DDL）和显示 SQL 结果集合的函数。

```
require 'java'
require 'lib/opencsv-2.3.jar'

TEMP_FILE="temp.csv"

def displayResultSet(r)
		writer = Java.AuComBytecodeOpencsv::CSVWriter.new(
			java.io.FileWriter.new(TEMP_FILE),
			java.lang.String.new("\t").charAt(0)
		)
		writer.writeAll(r, true)
		writer.close()
```

```
    File.open(TEMP_FILE).readlines.each{|line|puts line}
    `rm #{TEMP_FILE}`
end

def exec(statement, conn)
  puts statement
  conn.createStatement().execute(statement)
end

def execQuery(statement, conn)
  puts statement
  conn.createStatement().executeQuery(statement);
end
```

这个文件实际上是通过 `load 'dbutils.rb'` 加载到环境中的。

A.3.1 H2

Thomas Mueller 创造 H2（http://h2database.com）同时也大力参与了 HSQLDB 的开发。下面的例子使用了用户名 `sa` 和空密码来连接数据库 test，然后执行了 SQL 语句。同时需要注意的是，在创建表时没有指定 name 列的 VARCHAR 长度。

```
load 'dbutils.rb'

require 'jdbc/h2'
Jdbc::H2.load_driver

conn = java.sql.DriverManager.getConnection('jdbc:h2:test', "sa", "")
# VARCHAR does not require a length
exec("CREATE TABLE test (id int, name varchar)", conn)
exec("INSERT INTO test(id, name) VALUES (1, 'a')", conn)
displayResultSet(execQuery('select * from test', conn))
exec("DROP TABLE test", conn)
conn.close()
```

A.3.2 HSQLDB

HSQLDB（http://hsqldb.org/）因为被 OpenOffice 这样的开源项目以及 Mathematica 这样的商业项目所引用而出名。和 H2 不同，在获取连接时它不需要用户名和密码，而且 name 列使用的 VARCHAR 类型必须指定长度。

```
load 'dbutils.rb'

require 'jdbc/hsqldb'
Jdbc::HSQLDB.load_driver

con = java.sql.DriverManager.getConnection('jdbc:hsqldb:test')
exex("CREATE TABLE test (id, int, name varchar(10)", conn)
exec("INSERT INTO test(id, name) VALUES (1, 'a')", conn)
displayResultSet(execQuery('select * from test', conn)
```

```
exec("DROP TABLE test", conn)
conn.close()
```

A.3.3 Derby

Derby（http://db.apache.org/derby）这个项目的起源可以追溯到 20 世纪 90 年代。它随着 CloudScape、Informix 和 IBM 不断涅重生。从 Java 6 开始，Sun（后来被 Oracle 收购）将 Derby 作为 Java DB 引入 JDK。JDBC 连接字符串中的 `create=true` 属性表示在请求连接时如果没有该数据库就创建。此外，Derby 需要在删除引用表之前显式关闭其结果集：

```
load 'dbutils.rb'

require 'jdbc/derby'
Jdbc::Derby.load_driver

conn = java.sql.DriverManager.getConnection('jdbc:derby:test;create=true')
exec("CREATE TABLE test (id int, name varchar(10))", conn)
exec("INSERT INTO test(id, name) VALUES (1, 'a')", conn)
r = execQuery('select * from test', conn)
displayResultSet(r)
r.close()
exec("DROP TABLE test", conn)
conn.close()
```

虽然上一个例子聚焦于关系型数据库，但是显然任何 Java 库都可以通过脚本来访问，并且可以使用类似的步骤来探索。

A.4 小结

尽管整个环境已经开始走向图形化，但 CLI 提供的直接反馈输出还是令其成为一种特别高效的工具。我们可以通过在 CLI 环境中不断试验的方式来构造脚本，试验的过程不仅消除了传统的构建步骤，甚至不需要特地执行一个源文件。这种类型的交互对于很多开发者来说是易于理解的，但那些专注于 Java 的开发者也许只有有限的接触经验。现代 Web 浏览器里的 JavaScript 控制台就是一个现代的 CLI 实现，而且正如本章所阐述的，在服务器端也有类似的工具供 Java 程序员使用，只需要进行简单设置即可。

附录B

REST式的Web API总结

B.1　HTTP 1.1请求方法

表 B-1 总结了 HTTP 1.1 的请求方法（http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html）。

表B-1：HTTP 1.1请求方法

HTTP动词	对资源采取的动作	REST动作
`GET`	检索	类似 `SQL SELECT`
`HEAD`	不带响应体的检索	类似 `SQL SELECT 1`
`POST`	创建（或追加）	类似 `SQL INSERT`
`PUT`	更新（或创建）完整资源	类似 `SQL UPDATE`（或不存在时的 `INSERT`）
`PATCH`	部分更新	类似 `SQL UPDATE`（部分资源）
`DELETE`	删除	类似 `SQL DELETE`
`TRACE`	Echo 请求	确定中间服务器修改的诊断
`OPTIONS`	返回支持的方法	确定资源允许的 HTTP 方法
`CONNECT`	支持 HTTP 隧道	支持 HTTP 隧道技术

B.2　HTTP 1.1响应码

表 B-2 到表 B-6 总结了 HTTP 1.1 状态码（http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html）。

表B-2：信息类状态码1xx

状态码	意义	描述
100	继续	指示请求已经收到（而且没有被服务器拒绝）期间的中间响应
101	切换协议	服务器切换到由 UPgrade 请求头定义的协议

表B-3：成功状态码2xx

状态码	意义	描述
200	OK	被接受
201	被创建	正在创建一个新的资源
202	被接受	已接受，但是处理还未完成
203	非权威信息	实体头返回元数据的子集或超集
204	无内容	不包含响应体
205	重置内容	客户端应该发起一个请求来查看与初始请求相关联的资源
206	部分内容	一个包含了一个 range 请求头的响应

表B-4：重定向状态码3xx

状态码	意义	描述
300	多个选项	资源在不同位置有多种展现
301	永久移动	资源已经分配了一个新的永久 URI
302	找到	资源已经分配了一个新的临时 URI
303	参考其他	请求的响应在一个不同的 URI 下可用
304	未修改	文档未修改的条件请求的响应
305	使用代理	请求的资源可以通过返回的代理 URI 访问
306	（未使用）	当前的 HTTP 版本没有使用
307	临时重定向	请求的资源临时存在于一个不同的 URI 下

表B-5：客户端错误状态码4xx

状态码	意义	描述
400	坏请求	请求无法理解
401	未授权	请求没有授权
402	需要支付	保留以后使用
403	禁止	请求不允许（即使有附加授权）
404	未找到	资源未找到
405	方法不允许	指定 URL 的 HTTP 方法无效
406	不可接受	可以使用 accept 请求头中的内容来生成资源
407	需要代理验证	请求未验证（需要通过代理授权）
408	请求超时	客户端没有在服务器指定的时间里发送请求
409	冲突	因为当前资源的状态无法完成请求（例如，因为 PUT 所做的修改）
410	丢失	资源不可用
411	需要长度	需要内容长度头部

（续）

状态码	意义	描述
412	前置条件失败	请求头部的前置条件值为 `false`
413	请求实体太大	请求实体比服务器指定的阈值大
414	请求 URI 过长	请求 URI 比服务器指定的阈值长
415	不支持的媒体类型	格式不支持
416	请求范围不满足	头部指定的内容范围无法处理
417	期望失败	请求头部域的期望不满足

表B-6：服务器端错误状态码5xx

状态码	意义	描述
500	服务器内部错误	服务器未预期的错误情况
501	未实现	不支持的功能
502	坏网关	作为代理的服务器收到了来自上游服务器的无效响应
503	服务不可用	服务器临时不可用
504	网关超时	代理服务器未收到上游服务器的及时响应
505	HTTP 版本不支持	请求信息中的 HTTP 协议版本不支持

B.3 Curl Web API

Curl（http://curl.haxx.se/）实用工具可以使用各种不同协议与服务器传输数据。表 B-7 展示了使用该工具的命令行选项子集就可以完成的大部分 HTTP REST 式的 Web API 操作。

表B-7：精选的HTTP相关的Curl选项

选项	名称	描述
`-H`	头部	指定一个 HTTP 头部
`-d`	数据	发送指定的字符串数据到服务器
`-s`	静默选项	不显示进度表和错误信息
`-L`	位置	如果服务器响应带有位置头部和 3xx 响应状态码，则在新位置上重新发送请求（使用 `--max-redirs` 限制重定向）
`-X`	执行选项	指定 HTTP 请求方法
`-A`	代理	指定用户代理
`-b`	Cookie	指定 cookie（`--cookie` 比 `-b` 更容易记忆）
`-o`	输出	输出到一个文件（或者使用 `-O` 按照远程请求的名字写入一个文件）

调用示例：

```
curl -s -H "Accept: application/json" \
-H "Content-Type: application/json" \
http://localhost:8080/hello/world \
-X PUT -d '{"hello": "world"}'
```

B.4 JSON语法

JSON 是一种简单的数据交换格式，是 JavaScript 的子集。

JSON类型

- 数组（有序、方括号包裹的逗号隔开的值）
- 对象（无序、逗号隔开的键值对集合）
- 数字
- 字符串
- 布尔值
- Null

B.5 铁路图

下面的铁路图（http://bottlecaps.de/rr/ui）对组成 JSON 数据交换格式的 JavaScript 子集进行了更为正式的描述。

B.5.1 对象

JSON 对象是一组由花括号包裹的零个或多个字符串对及其关联值的集合。所有的字符串都跟着一个冒号，后面是其关联值。如果字符串值对不止一个，则会使用逗号隔开，如图 B-1 所示。

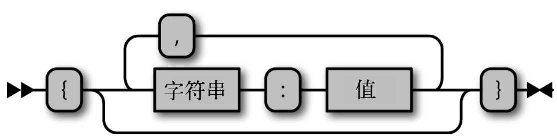


图 B-1：对象

B.5.2 数组

JSON 数组是由方括号包裹的由逗号隔开的值的列表，如图 B-2 所示。

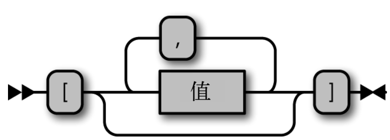


图 B-2：数组

B.5.3 值

值可以为字符串、数字、对象、数组、`true`、`false` 或者 `null`，如图 B-3 所示。

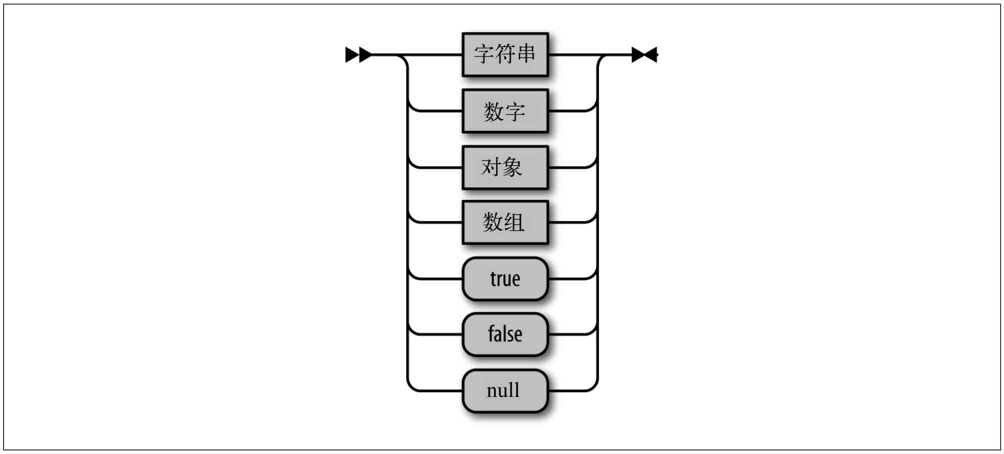


图 B-3：值

附 录 C

参考文献

Barrett, Daniel J. *Linux Pocket Guide, Essential Commands*. Sebastopol: O'Reilly Media, 2004.

Burke, Bill. *RESTful Java with JAX-RS*. Sebastopol: O'Reilly Media, 2009.

Crockford, Douglas. *JavaScript: The Good Parts*. Sebastopol: O'Reilly Media, 2008.

Fogus, Michael, and Chris Houser. *The Joy of Clojure*. Stamford: Manning Publications, 2011.

Hashimoto, Mitchell. *Vagrant: Up and Running*. Sebastopol: O'Reilly Media, 2013.

McConnell, Steve. *Code Complete: A Practical Handbook of Software Construction*. Redmond: Microsoft Press, 2004.

Pólya, George. *How to Solve It: A New Aspect of Mathematical Method*. Princeton: Princeton Science Library, 2004.

Resig, John and Bear Bibeault. *Secrets of the JavaScript Ninja*. Stamford: Manning Publications, 2012.

Sonatype Company. *Maven: The Definitive Guide*. Sebastopol: O'Reilly Media, 2008.

Thomas, Dave, with Chad Fowler and Andy Hunt. *Programming Ruby*. The Pragmatic Programmers, LLC, 2004.

Wynne, Matt, and Aslak Hellesøy. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. The Pragmatic Programmers, LLC, 2010.

Zakas, Nicholas Z. *Professional JavaScript for Web Developers*. Indianapolis: Wrox Press, 2005.

关于作者

Casimir Saternos 有十余年软件开发经验，曾为 *Java Magazine* 和 Oracle Technology Network 撰写技术文章，参与录制了几期 Peepcode screencasts。现在，他主要使用 Java、Ruby 等技术从事 Web 应用开发。

关于封面图

本书封面上的动物是印度灵猫（Viverra zibetha）。这种哺乳动物分布于东南亚的草原和密林，其中包括缅甸、泰国、柬埔寨、马来西亚和中国南方地区。

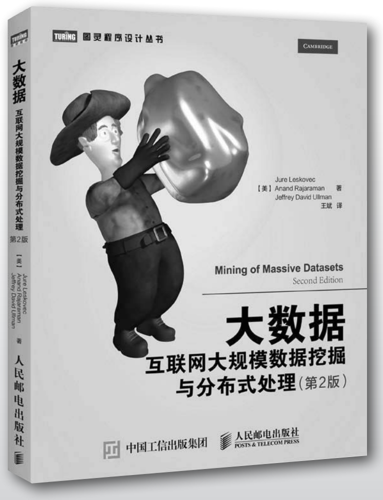
大印度灵猫喜独居，夜间最为活跃，喜食小鸟、蛇、青蛙以及相对较小的哺乳动物。它也食用水果和树根，但以肉食为主。印度灵猫白天宿于洞穴中（常常是其他动物挖掘和废弃不用的洞穴）。

大印度灵猫身长 20~37 英寸（不包括尾巴的长度）。它的毛皮呈灰褐色，颈部和尾部有黑色条纹。雌性灵猫比雄性灵猫略小，可随时交配（一般一年生两胎，独自抚养后代）。

灵猫香是灵猫的排泄物，用于标识领地。稀释后的灵猫香（通常采集自非洲灵猫）几个世纪以来都被用作香水的定香剂。很多现代产品中已经开始使用人工合成的灵猫香，但是还有几种灵猫被非法捕捉以获取其肉和臭腺。

封面图片来自 Lydekker 的 *Natural History*。

图灵最新重点图书

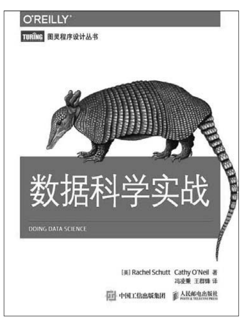


大数据权威著作全新升级版！

第 1 版畅销 40000 册！

本书源自作者在斯坦福大学教授的“海量数据挖掘”（CS246: Mining Massive Datasets）课程，第 1 版上市以来受到读者广泛欢迎和认可。本书以大数据环境下的数据挖掘和机器学习为重点，全面介绍了实践中行之有效的数据处理算法，是在校学生和相关从业人员的必备读物。

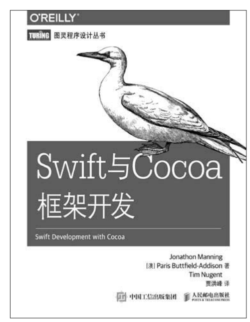
大数据（第 2 版）
书号：978-7-115-39525-2
定价：79.00 元



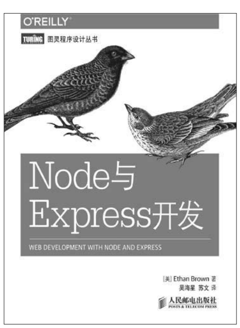
数据科学实战
书号：978-7-115-38349-5
定价：79.00 元



命令行中的数据科学
书号：978-7-115-39168-1
定价：49.00 元



Swift 与 Cocoa 框架开发
书号：978-7-115-39187-2
定价：89.00 元



Node 与 Express 开发
书号：978-7-115-38033-3
定价：69.00 元



Java 8 函数式编程
书号：978-7-115-38488-1
定价：39.00 元



学习响应式设计
书号：978-7-115-38973-2
定价：69.00 元

O'REILLY®

全端Web开发：使用JavaScript与Java

近几年来，用户习惯、技术和开发方法极大地改变了Web应用的设计，但是Web本身并没有变化。本书展示了如何开发出遵循Web底层架构的应用。

作为一名Java程序员，在以客户端－服务器端架构开发Web应用时，如何应对各种难题？这本内容详尽的指南将告诉你如何使用各种Java工具、客户端技术和Web API开发Web应用。作者首先概括了客户端－服务器端技术，然后详细介绍了很多实用的客户端－服务器端架构。你将在多个章节中参与到实战项目中，从而获得对相应技术和主题的第一手经验。

通过阅读本书，你将会：

- **了解客户端和服务器端分层的好处，包括代码组织和快速原型开发；**
- **探索JavaScript开发中用到的各种主流工具、框架和起点项目；**
- **深入学习Web API设计和REST风格的软件架构；**
- **了解有别于传统打包方法的各种Java打包方式，以及应用服务器的部署；**
- **使用轻量级服务器构建项目，涉及jQuery和Jython、Sinatra和Angular；**
- **使用传统Java Web应用服务器和类库构建客户端－服务器端Web应用。**

“随着客户端－服务器端架构向浏览器迁移，现在的程序员面临着来自新技术和架构的挑战。这本书直捣这一复杂性的核心，将Web应用开发的现状直接呈现在读者眼前。”

——Tony Powell

Trifecta Technologies公司的技术方案负责人

Casimir Staternos是Synchronoss Technologies公司软件架构师，有十余年软件开发经验。曾在*Java Magazine*和Oracle Technology Network上发表过技术文章，可在Pluralsight（www.pluralsight.com）上观看他录制的Peepcode播客视频。他目前主要使用Java、Ruby等技术从事Web应用开发。

JAVASCRIPT

封面设计：**Karen Montgomery** 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机/Web开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-39730-0
定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们:

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks